Revit 2008 API

# User Manual

Version 1.0

**April, 2007**

## Copyright. 2007 Autodesk, Inc.

All Rights Reserved

This publication, or parts thereof, may not be reproduced in any form, by any method, for any purpose.

**AUTODESK, INC., MAKES NO WARRANTY, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE REGARDING THESE MATERIALS, AND MAKES SUCH MATERIALS AVAILABLE SOLELY ON AN "AS-IS" BASIS.**

**IN NO EVENT SHALL AUTODESK, INC., BE LIABLE TO ANYONE FOR SPECIAL, COLLATERAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES IN CONNECTION WITH OR ARISING OUT OF ACQUISITION OR USE OF THESE MATERIALS. THE SOLE AND EXCLUSIVE LIABILITY TO AUTODESK, INC., REGARDLESS OF THE FORM OF ACTION, SHALL NOT EXCEED THE PURCHASE PRICE, IF ANY, OF THE MATERIALS DESCRIBED HEREIN.**

Autodesk, Inc., reserves the right to revise and improve its products as it sees fit. This publication describes the state of the product at the time of publication, and may not reflect the product at all times in the future.

## Autodesk Trademarks

The following are registered trademarks or trademarks of Autodesk, Inc., in the USA and other countries: 3DEC (design/logo), 3December, 3December.com, 3ds Max, ActiveShapes, Actrix, ADI, Alias, Alias (swirl design/logo), AliasStudio, Alias|Wavefront (design/logo), ATC, AUGI, AutoCAD, AutoCAD Learning Assistance, AutoCAD LT, AutoCAD Simulator, AutoCAD SQL Extension, AutoCAD SQL Interface, Autodesk, Autodesk Envision, Autodesk Insight, Autodesk Intent, Autodesk Inventor, Autodesk Map, Autodesk MapGuide, Autodesk Streamline, AutoLISP, AutoSnap, AutoSketch, AutoTrack, Backdraft, Built with ObjectARX(logo), Burn, Buzzsaw, CAiCE, Can You Imagine, Character Studio, Cinestream, Civil 3D, Cleaner, Cleaner Central, ClearScale, Colour Warper, Combustion, Communication Specification, Constructware, Content Explorer, Create>what's>Next> (design/logo), Dancing Baby (image), DesignCenter, Design Doctor, Designer's Toolkit, DesignKids, DesignProf, DesignServer, DesignStudio, Design|Studio (design/logo), Design Your World, Design Your World (design/logo), DWF, DWG, DWG (logo), DWG TrueConvert, DWG TrueView, DXF, EditDV, Education by Design, Extending the Design Team, FBX, Filmbox, FMDesktop, GDX Driver, Gmax, Heads-up Design, Heidi, HOOPS, HumanIK, i-drop, iMOUT, Incinerator, IntroDV, Kaydara, Kaydara (design/logo), LocationLogic, Lustre, Maya, Mechanical Desktop, MotionBuilder, ObjectARX, ObjectDBX, Open Reality, PolarSnap, PortfolioWall, Powered with Autodesk Technology, Productstream, ProjectPoint, Reactor, RealDWG, Real-time Roto, Render Queue, Revit, Showcase, SketchBook, StudioTools, Topobase, Toxik, Visual, Visual Bridge, Visual Construction, Visual Drainage, Visual Hydro, Visual Landscape, Visual Roads, Visual Survey, Visual Syllabus, Visual Toolbox, Visual Tugboat, Visual LISP, Voice Reality, Volo, and Wiretap.

The following are registered trademarks or trademarks of Autodesk Canada Co. in the USA and/or Canada and other countries: Backburner, Discreet, Fire, Flame, Flint, Frost, Inferno, Multi-Master Editing, River, Smoke, Sparks, Stone, Wire.

## Third Party Trademarks

All other brand names, product names or trademarks belong to their respective holders.

## Third Party Software Program Credits

ACIS Copyright. 1989-2001 Spatial Corp. Portions Copyright. 2002 Autodesk, Inc.

Copyright. 1997 Microsoft Corporation. All rights reserved.

Flash □ is a registered trademark of Macromedia, Inc. in the United States and/or other countries.

International CorrectSpell™ Spelling Correction System. 1995 by Lernout & Hauspie Speech Products, N.V. All rights reserved.

InstallShield™ 3.0. Copyright. 1997 InstallShield Software Corporation. All rights reserved.

PANTONE□ Colors displayed in the software application or in the user documentation may not match PANTONE-identified standards. Consult current PANTONE Color Publications for accurate color.

PANTONE□ and other Pantone, Inc. trademarks are the property of Pantone, Inc.. Pantone, Inc., 2002

Pantone, Inc. is the copyright owner of color data and/or software which are licensed to Autodesk, Inc., to distribute for use only in combination with certain Autodesk software products. PANTONE Color Data and/or Software shall not be copied onto another disk or into memory unless as part of the execution of this Autodesk software product.

Portions Copyright. 1991-1996 Arthur D. Applegate. All rights reserved.

Portions of this software are based on the work of the Independent JPEG Group.

RAL DESIGN. RAL, Sankt Augustin, 2002

RAL CLASSIC. RAL, Sankt Augustin, 2002

Representation of the RAL Colors is done with the approval of RAL Deutsches Institut für Gütesicherung und Kennzeichnung e.V. (RAL German Institute for Quality Assurance and Certification, re. Assoc.), D-53757 Sankt Augustin.

Typefaces from the Bitstream□ typeface library copyright 1992.

Typefaces from Payne Loving Trust. 1996. All rights reserved.

AutoCAD 2008 is produced under a license of data derived from DIC Color Guide□ from Dainippon Ink and Chemicals, Inc. Copyright . Dainippon Ink and Chemicals, Inc. All rights reserved. DIC Color Guide computer color simulations used in this product may not exactly match DIC Color Guide, DIC color Guide Part 2 identified solid color standards. Use current DIC Color Guide Manuals for exact color reference. DIC and DIC Color Guide are registered trademarks of Dainippon Ink and Chemicals, Inc.

Printed manual and help produced with Idiom WorldServer™.

WindowBlinds: DirectSkin™ OCX . Stardock®

AnswerWorks 4.0 .; 1997-2003 WexTech Systems, Inc. Portions of this software . Vantage-Knexys. All rights reserved.

The Director General of the Geographic Survey Institute has issued the approval for the coordinates exchange numbered TKY2JGD for Japan Geodetic Datum 2000, also known as technical information No H1-N0.2 of the Geographic Survey Institute, to be installed and used within this software product (Approval No.: 646 issued by GSI, April 8, 2002).

Portions of this computer program are copyright . 1995-1999 LizardTech, Inc. All rights reserved. MrSID is protected by U.S. Patent No. 5,710,835. Foreign Patents Pending.

Portions of this computer program are Copyright .; 2000 Earth Resource Mapping, Inc.

OSTN97 . Crown Copyright 1997. All rights reserved.

OSTN02 . Crown copyright 2002. All rights reserved.

OSGM02 . Crown copyright 2002, . Ordnance Survey Ireland, 2002.

FME Objects Engine . 2005 SAFE Software. All rights reserved.

GOVERNMENT USE

Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in FAR 12.212 (Commercial Computer Software-Restricted Rights) and DFAR 227.7202 (Rights in Technical Data and Computer Software), as applicable.

# Table of Contents


**1 Welcome to the Revit API ..... 13**

1.1 Introduction to the Revit API ..... 13

1.2 What Can You Do with the Revit API? ..... 13

1.3 Requirements ..... 13

1.4 Install and Learn the Revit-Based Product ..... 14

1.5 Installation ..... 14

1.6 Supported Programming Languages ..... 14

1.7 User Manual ..... 14

1.7.1 Introduction to the Revit API ..... 15

1.7.2 Basic Topics ..... 15

1.7.3 Element Topics ..... 15

1.7.4 Advanced Topics ..... 16

1.7.5 Product Specific ..... 16

1.7.6 Other ..... 16

1.8 Documentation Conventions ..... 16

1.9 Object Model Diagram ..... 16

1.10 What's new in this release ..... 17

**2 Getting Started ..... 19**

2.1 Walkthroughs ..... 19

2.2 Walkthrough: Hello World ..... 19

2.2.1 Create a New Project ..... 19

2.2.2 Add References ..... 20

2.2.3 Change the Class Name ..... 22

2.2.4 Add Code ..... 22

2.2.5 Build the Program ..... 23

2.2.6 Modify the Revit.ini file ..... 23

2.2.7 Debug the Add-in ..... 23

2.2.8 Troubleshooting ..... 25

2.3 Walkthrough: Add Hello World Menu ..... 26

2.3.1 Create a New Project ..... 26

2.3.2 Change the Class Name ..... 26

2.3.3 Add Code ..... 26

2.3.4 Build the Application ..... 27

2.3.5 Modify the Revit.ini File ..... 27

2.3.6 Debugging ..... 27

2.4 Walkthrough: Retrieve Selected Elements ..... 28

2.5 Walkthrough: Retrieve All Elements ..... 29
**3 Add-In Integration ..... 31**
3.1 Overview ..... 31
3.2 External Command ..... 31
3.2.1 Mechanism ..... 32
3.2.2 IExternalCommand ..... 33
3.2.3 [ExternalCommands] ..... 37
3.3 External Application ..... 38
3.3.1 IExternalApplication ..... 38
3.3.2 [ExternalApplications] ..... 40
3.4 Access to Revit Events ..... 40
3.4.1 OnDialogBox ..... 42
3.4.2 OnDocumentClosed ..... 42
3.4.3 OnDocumentNewed ..... 43
3.4.4 OnDocumentOpened ..... 43
3.4.5 OnDocumentSaved ..... 43
3.4.6 OnDocumentSavedAs ..... 43
3.5 Menu Item and Toolbar ..... 43
3.5.1 Create a New TopMenu and Menu Items ..... 43
3.5.2 Create a New Toolbar ..... 45
3.6 APIObject ..... 46
3.6.1 APIObject.IsReadOnly Property ..... 46
3.6.2 APIObject.ToString() Method ..... 47
3.6.3 Equals ..... 47
**4 Application and Document ..... 48**
4.1 Application Functions ..... 48
4.1.1 Application Version Information ..... 48
4.1.2 Application-Wide Settings ..... 48
4.1.3 Document Management ..... 48
4.1.4 Shared Parameter Management ..... 49
4.1.5 Menu and Toolbar Utility ..... 49
4.1.6 Event ..... 49
4.1.7 Create ..... 49
4.2 Document Functions ..... 49
4.2.1 Settings Property ..... 49
4.2.2 Place and Locations ..... 49
4.2.3 Type Collections ..... 49
4.2.4 View Management ..... 49

4.2.5 Element Retrieval ..... 50
4.2.6 File Management ..... 50
4.2.7 Element Management ..... 50
4.2.8 Transactions ..... 50
4.2.9 Others ..... 51
4.3 Document and File Management ..... 51
4.3.1 Document Retrieval ..... 51
4.3.2 Document File Information ..... 51
4.3.3 Open a Document ..... 51
4.3.4 Create a Document ..... 52
4.3.5 Save and Close a Document ..... 52
4.3.6 Load Family and Load Family Symbol ..... 52
4.4 Settings ..... 52
4.5 Unit ..... 55
**5 Elements Essentials ..... 57**
5.1 Elements Classification ..... 57
5.1.1 Model Elements ..... 57
5.1.2 View Elements ..... 58
5.1.3 Group Elements ..... 59
5.1.4 Annotation and Datum Elements ..... 59
5.1.5 Sketch Elements ..... 61
5.1.6 Information Elements ..... 61
5.2 Other Classifications ..... 62
5.2.1 Categories ..... 62
5.2.2 Family ..... 65
5.2.3 Symbol ..... 66
5.2.4 Instance ..... 67
5.3 Element Retrieval ..... 67
5.3.1 General ..... 67
5.3.2 Selection ..... 68
5.3.3 Other ..... 69
5.4 General Properties ..... 69
5.4.1 ElementId ..... 69
5.4.2 UniqueId ..... 70
5.4.3 Location ..... 70
5.4.4 Level ..... 72
5.4.5 Parameter ..... 72
**6 Parameter ..... 75**

6.1 Walkthrough: Get Selected Element Parameters ..... 75
6.2 Definition ..... 77
6.2.1 ParameterType ..... 78
6.2.2 ParameterGroup ..... 78
6.3 BuiltInParameter ..... 78
6.4 StorageType ..... 79
6.5 AsValueString and SetValueString ..... 81
6.6 Parameter Tips and Tricks ..... 82
6.7 Add Parameters to Elements ..... 84
**7 Collection ..... 85**
7.1 Interface ..... 85
7.1.1 IEnumerable ..... 85
7.1.2 IEnumerator ..... 86
7.1.3 As and Is Operators ..... 86
7.2 Collections and Iterators ..... 87
**8 Editing Elements ..... 91**
8.1 Move ..... 91
8.2 Rotate ..... 93
8.3 Delete ..... 95
8.4 Mirror ..... 96
8.5 Group ..... 97
8.6 Array ..... 100
8.7 SuspendUpdating ..... 100
**9 Host Elements and Opening ..... 103**
9.1 Wall ..... 103
9.2 Floor and Foundation ..... 105
9.3 Other Elements ..... 108
9.3.1 Stair and Ramp ..... 108
9.3.2 Ceiling and Roof ..... 108
9.4 CompoundStructure ..... 108
9.4.1 Material ..... 110
9.5 Opening ..... 111
9.5.1 General Properties ..... 112
9.5.2 Create Opening ..... 113
**10 Component Elements ..... 117**
10.1 Identifying Elements ..... 117
10.2 Family ..... 119
10.2.1 Components ..... 119

10.2.2 Solid and Void Forms ..... 120
10.2.3 Others ..... 121
10.2.4 Load ..... 122
10.3 FamilyInstance ..... 122
10.3.1 Location-Related Properties ..... 122
10.3.2 Host and Component ..... 125
10.3.3 Other Properties ..... 126
10.3.4 Creation ..... 126
10.4 Code Samples ..... 127
10.4.1 Create Tables ..... 127
10.4.2 Create a Beam ..... 128
10.4.3 Create Doors ..... 129
10.4.4 Create FamilyInstances Using Reference Directions ..... 130
**11 Datum and Information Elements ..... 133**
11.1 Level ..... 134
11.1.1 Elevation ..... 134
11.1.2 Create a Level ..... 136
11.2 Grid ..... 137
11.2.1 Curve ..... 137
11.2.2 Create a Grid ..... 138
11.3 Phase ..... 139
11.4 Design Options ..... 141
**12 Annotation Elements ..... 143**
12.1 Dimension and Constraint ..... 144
12.1.1 Permanent Dimensions ..... 144
12.1.2 Constraint Elements ..... 145
12.1.3 Spot Dimensions ..... 147
12.1.4 Comparison ..... 147
12.1.5 Create and Delete ..... 148
12.2 Detail Curve ..... 148
12.3 Tags ..... 149
12.4 Text ..... 151
12.5 Annotation Symbol ..... 152
12.5.1 Create and Delete ..... 152
12.5.2 Add and Remove Leader ..... 152
**13 Sketching ..... 153**
13.1 2D Sketch Class ..... 153
13.2 3D Sketch ..... 156

13.2.1 Extrusion ..... 157
13.2.2 Revolution ..... 158
13.2.3 Blend ..... 160
13.2.4 Sweep ..... 162
13.3 ModelCurve ..... 164
13.3.1 Create ..... 165
13.3.2 Properties ..... 166
**14 Views ..... 169**
14.1 Overview ..... 169
14.1.1 View Process ..... 169
14.1.2 View Types ..... 170
14.1.3 Element Visibility in a View ..... 173
14.1.4 Create and Delete Views ..... 174
14.2 View3D ..... 175
14.2.1 Perspective View ..... 176
14.2.2 Orthographic View ..... 178
14.2.3 3D Views SectionBox ..... 179
14.3 ViewPlan ..... 181
14.4 ViewDrafting ..... 182
14.5 ViewSection ..... 182
14.6 ViewSheet ..... 184
**15 Material ..... 187**
15.1 General Material Information ..... 187
15.1.1 Classification ..... 187
15.1.2 Properties ..... 190
15.2 Material Management ..... 192
15.2.1 Create ..... 192
15.2.2 Delete ..... 193
15.3 Element Material ..... 194
15.3.1 Material in a Parameter ..... 194
15.3.2 Material and Category ..... 196
15.3.3 CompoundStructureLayer Material ..... 197
15.3.4 Retrieve Element Materials ..... 198
15.3.5 Walkthrough: Get Window Materials ..... 199
**16 Geometry ..... 201**
16.1 Example: Retrieve Geometry Data from a Wall ..... 201
16.1.1 Create Geometry Options ..... 201
16.1.2 Retrieve Faces and Edges ..... 202

16.2 Geometry Node Class ..... 202
16.2.1 Geometry.Instance ..... 203
16.2.2 Geometry.Mesh ..... 205
16.2.3 Solid ..... 206
16.3 Geometry Helper Class ..... 207
16.3.1 Geometry.Transform ..... 207
16.3.2 Geometry.Reference ..... 210
16.3.3 Geometry.Options ..... 212
16.3.4 Geometry.BoundingBoxXYZ ..... 214
16.3.5 Geometry.BoundingBoxUV ..... 218
16.4 Collection Classes ..... 218
16.5 Example: Retrieve Geometry Data from a Beam ..... 219
**17 Place and Locations ..... 223**
17.1 Place ..... 223
17.2 City ..... 224
17.3 ProjectLocation ..... 225
17.4 Project Position ..... 225
17.4.1 Create and Delete ..... 228
**18 Shared Parameter ..... 231**
18.1 Definition File ..... 231
18.1.1 Definition File Format ..... 231
18.2 Definition File Access ..... 233
18.2.1 Create a Shared Parameter File ..... 235
18.2.2 Access an Existing Shared Parameter File ..... 235
18.2.3 Change the Parameter Definition Owner Group ..... 236
18.3 Binding ..... 237
18.3.1 Type Binding ..... 238
18.3.2 Instance Binding ..... 239
**19 Transaction ..... 243**
19.1 Usage ..... 243
19.1.1 Atomic User Actions ..... 243
19.1.2 Get Created Element Geometry and AnalyticalModel ..... 245
19.2 Boundaries ..... 246
**20 Revit Architecture ..... 249**
20.1 Room ..... 249
20.1.1 Room, Area, and Tags ..... 249
20.1.2 Create ..... 250
20.1.3 Room Boundary ..... 250

20.1.4 Plan Topology ..... 255
20.1.5 Room and FamilyInstance ..... 257
20.2 Energy Data ..... 259
**21 Revit Structure ..... 261**
21.1 Structural Model Elements ..... 261
21.1.1 Column, Beam, and Brace ..... 261
21.1.2 AreaReinforcement and PathReinforcement: ..... 262
21.1.3 BeamSystem ..... 265
21.1.4 Rebar ..... 265
21.1.5 BoundaryConditions ..... 267
21.1.6 Other Structural Elements ..... 268
21.2 AnalyticalModel ..... 269
21.2.2 RigidLink ..... 271
21.2.3 SupportData ..... 272
21.3 Loads ..... 275
21.3.1 Load Settings ..... 275
21.3.2 Load Limitations ..... 276
21.4 Analysis Link ..... 276
**A Glossary ..... 279**
A.1 API ..... 279
A.2 Array ..... 279
A.3 BIM ..... 279
A.4 Class ..... 279
A.5 Events ..... 279
A.6 GUI ..... 279
A.7 Iterator ..... 279
A.8 Method ..... 279
A.9 Namespace ..... 279
A.10 Overloading ..... 280
A.11 Properties ..... 280
A.12 Revit Families ..... 280
A.13 Revit Parameters ..... 280
A.14 Revit Types ..... 280
A.15 Sets ..... 280
A.16 Element ID ..... 280
A.17 Element UID ..... 280
**B FAQ ..... 281**
B.1 General Questions ..... 281

B.2 Revit Structure Questions ..... 282
**C Hello World for VB.NET ..... 285**
C.1 Create a New Project ..... 285
C.2 Add Reference and Namespace ..... 285
C.3 Change the Class Name ..... 286
C.4 Add Code ..... 287
C.5 Modify the Revit.ini File ..... 287
C.6 Build the Program ..... 288
C.7 Debug the Program ..... 288

# 1 Welcome to the Revit API

All Revit-based products are Parametric Building Information Modeling (BIM) tools. These tools are similar to Computer-Aided Design (CAD) programs but are used to build 3D models rather than create drawings. In Revit, you place real-world elements like columns and walls into the model. Once the model is built, you can create model views such as sections and callouts. Views are generated from the 3D physical model; consequently, changes made in one view automatically propagate through all views. This virtually eliminates the need to update multiple drawings and details when you make changes to the model.

## 1.1 Introduction to the Revit API

The Revit .NET API allows you to program with any .NET compliant language including Visual Basic.NET, C#, and C++/CLI. In addition, the API is focused on integrating analysis and visualization applications into the Revit Building Information Model.

Revit Architecture 2008, Revit Structure 2008, and Revit MEP 2008 all contain the Revit API so that you can integrate your applications into Revit. The three APIs are very similar and are jointly referred to as the Revit 2008 API. Before using the API, learn to use Revit and its features so that you can better understand the relevant areas related to your programming. Learning Revit can help you:

1. Maintain consistency with the Revit UI and commands.
2. Design your add-in application seamlessly.
3. Master API classes and class members efficiently and effectively.

If you are not familiar with Revit or BIM, learn more in the Revit product center at http://www.autodesk.com/revit.

## 1.2 What Can You Do with the Revit API?

You can use the Revit API to:

- Gain access to model graphical data.
- Gain access to model parameter data.
- Create, edit, and delete model elements like floors, walls, columns, and more.
- Create add-ins to automate repetitive tasks.
- Integrate applications into Revit-based vertical products. Examples include linking an external relational database to Revit or sending model data to an analysis application.
- Perform analysis of all sorts using BIM.
- Automatically create project documentation.

## 1.3 Requirements

To go through the user manual, you need the following:

1. A working understanding of Revit Architecture 2008, Revit Structure 2008, and Revit MEP 2008.
2. You must know at least one Common Language Specification compliant language. For more information about CLS-compliant languages, review the information at the following link: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconwhatiscommonlanguagespecification.asp

3. Microsoft Visual Studio 2005 or Microsoft Visual Studio 2005 Express Edition.
4. Microsoft .NET Framework 2.0 if not already installed with Visual Studio.
5. The Revit Software Developer's Kit (SDK) which you can download from the Autodesk Developer Network (ADN) or the Revit installation CD/DVD (<DVD_Drive>:\COMMON\Revit SDK).
6. Two days of free time.

## 1.4 Install and Learn the Revit-Based Product

The Revit API requires the Microsoft .NET Framework 2.0 SDK. The examples in this document use Microsoft Visual Studio 2005 for the Integrated Development Environment (IDE).

If you are a beginning programmer, learn Microsoft Visual Studio 2005 and a compatible language such as C# or Visual Basic.Net. There are many good books and classes to get you started.

If you are a Revit novice, go through the tutorials which are accessible from the Revit Help menu. If possible, take a training class from your local Autodesk reseller to help you quickly get up to speed.

Regardless of your experience level, you can join one of the many discussion groups dedicated to Revit and the Revit API. The following resource links are a good starting point.

www.autodesk.com/revitbuilding

www.autodesk.com/revitstructure

www.autodesk.com/revitsystems

www.autodesk.com/bim

www.revitcity.com

www.augi.com

http://discussion.autodesk.com

Select = Autodesk Revit Architecture

Then Select = Autodesk Revit API

http://forums.augi.com/forumdisplay.php?f=93

http://discussion.autodesk.com/index2.jspa?categoryID=27

http://discussion.autodesk.com/index2.jspa?categoryID=104

http://discussion.autodesk.com/index2.jspa?categoryID=114

## 1.5 Installation

The Revit API is installed with Revit Architecture, Revit Structure, and Revit MEP. Any .NET based application will reference the RevitAPI.dll located in the Revit Program directory.

## 1.6 Supported Programming Languages

The Revit API is fully accessible by any language compatible with the Microsoft .NET Framework 2.0, such as Visual Basic .NET or Visual C#.

## 1.7 User Manual

This document is part of the Revit SDK. It provides an introduction to implementing Revit add-in applications using the Revit API.

Before creating a Revit API add-in application read through the manual and try the sample code. If you already have some experience with the Revit API, you may just want to review the Notes and Troubleshooting sections.

### 1.7.1 Introduction to the Revit API

The first two chapters present an introduction to the Revit API and provide an overview of the User Manual.

- Welcome to the Revit API - Presents an introduction to the Revit API and necessary prerequisite knowledge before you create your first add-in.
- Getting Started - Step-by-step instructions for creating your first Hello World add-in application using Visual Studio 2005 and three other walkthroughs covering primary add-in functions.

### 1.7.2 Basic Topics

These chapters cover the Revit API basic mechanisms and functionality.

- Add-in Integration - Discusses how an add-in is integrated into the Revit UI and invoked by user commands or specific Revit events such as program startup.
- Application and Document - Application and Document classes respectively represent the Revit application and project file in the Revit API. This chapter explains basic concepts and links to pertinent chapters and sections.
- Elements Essentials - The bulk of the data in a Revit project is in a collection of Elements. This chapter discusses the essential Element mechanism, classification, and features.
- Parameter - Most Element information is stored as Parameters. This chapter discusses Parameter functionality.
- Collection - Utility collection types such as Array, Map, Set collections, and related Iterators.

### 1.7.3 Element Topics

Elements are introduced based on element classification. Make sure that you read the Elements Essentials and Parameter chapters before reading about the individual elements.

- Editing Elements – Learn how to move, rotate, delete, mirror, group, and array one element or a set of elements.
- Host Elements and Opening – Discusses Elements, their corresponding Symbols representing built-in place construction, and different types of Openings in the API.
- Component Elements – Component Elements are Family Instance elements belonging to the Model Element. Learn about the relationship between family and family instance, family and family instance features, and how to load or create them.
- Datum and Information Elements – Learn how to set up grids, add levels, use Design Options, and more.
- Annotation Elements – Discusses document annotation including adding dimensions, detail curves, tags, and annotation symbols.
- Sketching - Sketch functions include 2D and 3D sketch classes such as SketchPlane, ModelCurve, GenericForm, and more.
- Views – Learn about the different ways to view models and components and how to manipulate the view in the API.
- Material - Material data is an Element that identifies the physical materials used in the project as well as texture, color, and more.

### 1.7.4 Advanced Topics

Before reviewing the advanced topics, be sure to review the basic and Element topics.

- Geometry – Discusses graphics-related types in the API used to describe the graphical representation of the model including the three classes that describe and store the geometry information.
- Place and Locations – Defines the project location including city, country, latitude, and longitude.
- Shared Parameter – Shared parameters are external text files containing parameter information. This chapter introduces how to gain access to shared parameters through the Revit API.
- Transaction –Introduces the two uses for Transaction and the limits that you must consider when using Transaction.

### 1.7.5 Product Specific

Revit products include Revit Architecture, Revit Structure, and Revit MEP. Some APIs only work in specific products.

- Revit Architecture - Discusses the APIs specific to Revit Architecture.
- Revit Structure - Discusses the APIs specific to Revit Structure.

### 1.7.6 Other

- Glossary – Definitions of terms used in this document..
- Appendix – Additional information such as Frequently Asked Questions, Using Visual Basic.Net for programming, and more.

## 1.8 Documentation Conventions

This document contains class names in namespace format, such as Autodesk.Revit.Element. In C++/CLI Autodesk.Revit.Element is Autodesk::Revit::Element. Since Visual C# is commonly used in this manual, the default namespace is Autodesk.Revit.

## 1.9 Object Model Diagram

The following picture shows the Revit 2008 API Object Model Diagram. For more information, review the Revit API Help system.

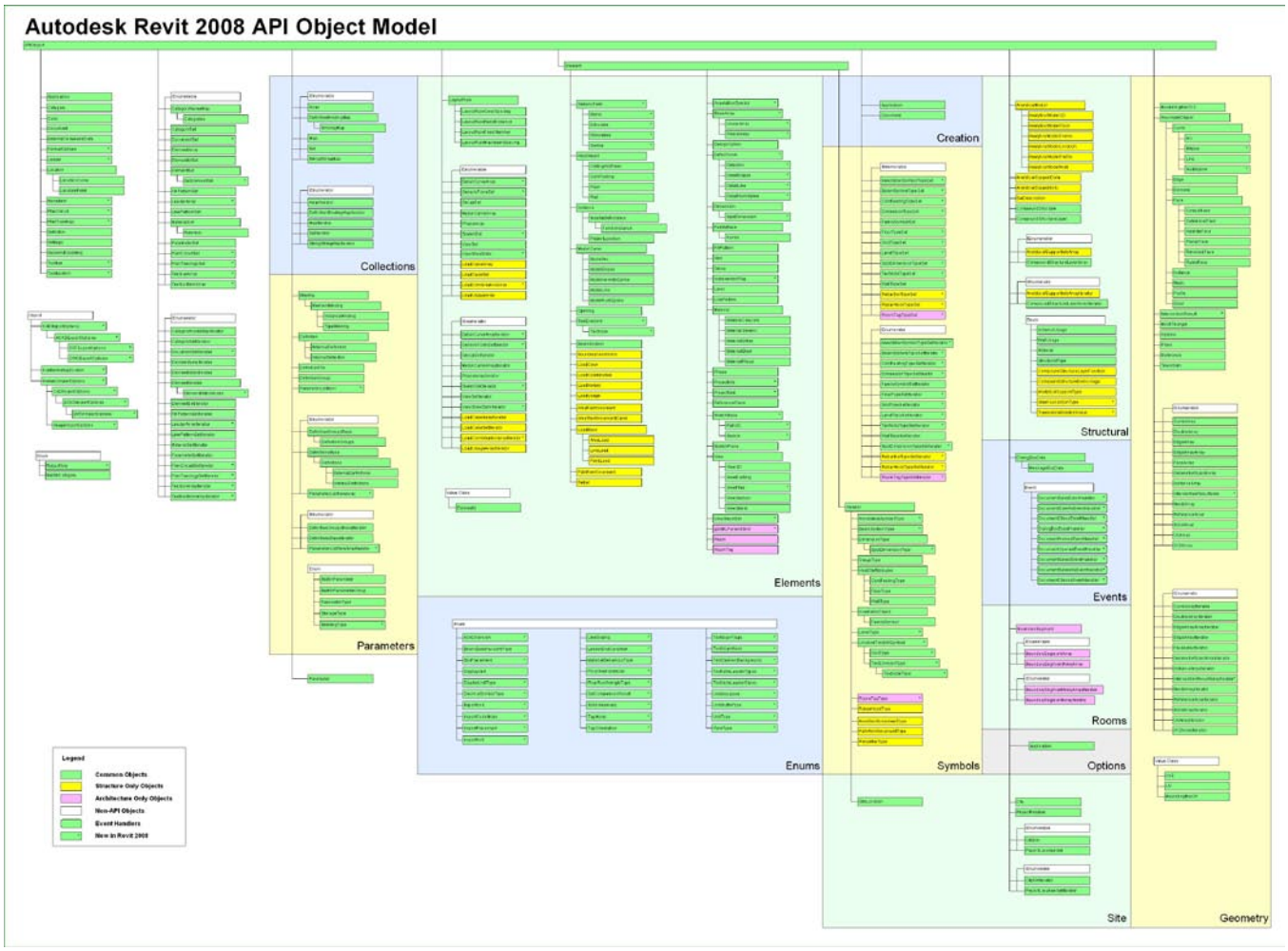



**Figure 1: Revit 2008 API Object Model diagram**

## 1.10 What's new in this release

Please see the documents that are available in the Revit 2008 SDK for information on changes from the previous version.

# 2 Getting Started

The Revit API is fully accessible by any language compatible with the Microsoft .NET Framework 2.0, such as Visual C# or Visual Basic .NET (VB.NET). Both Visual C# and VB.NET are commonly used to develop Revit API applications. However, the focus of this manual is developing applications using Visual C#.

## 2.1 Walkthroughs

If you are new to the Revit API, the following topics are good starting points to help you understand the product. Walkthroughs provide step-by-step instructions for common scenarios, helping you learn about the product or a particular feature. The following walkthroughs will help you get started using the Revit API:

- Walkthrough: Hello world - Illustrates how to create an add-in using the Revit API.
- Walkthrough: Add Hello World Menu - Illustrates how to add a menu item.
- Walkthrough: Retrieve Selected Elements - Illustrates how to retrieve selected elements.
- Walkthrough: Retrieve All Elements - Illustrates how to retrieve all document elements.

## 2.2 Walkthrough: Hello World

Use the Revit API and C# to create a Hello World program using the directions provided. For information about how to create an add-in application using VB.NET, refer to the section Hello World for VB.NET in the Appendix.

The Hello World walkthrough covers the following topics:

- Create a new project.
- Add references.
- Change the class name.
- Write the code
- Debug the add-in.

All operations and code in this section were created using Visual Studio 2005.

### 2.2.1 Create a New Project

The first step in writing a C# program with Visual Studio is to choose a project type and create a new Class Library.

1. From the File menu, select New> Project….
2. In the Project Types frame, click Visual C#.
3. In the Templates frame, click Class Library. This walkthrough assumes that the project location is: D:\Sample.
4. In the Name field, type HelloWorld as the project name.
5. Click OK.

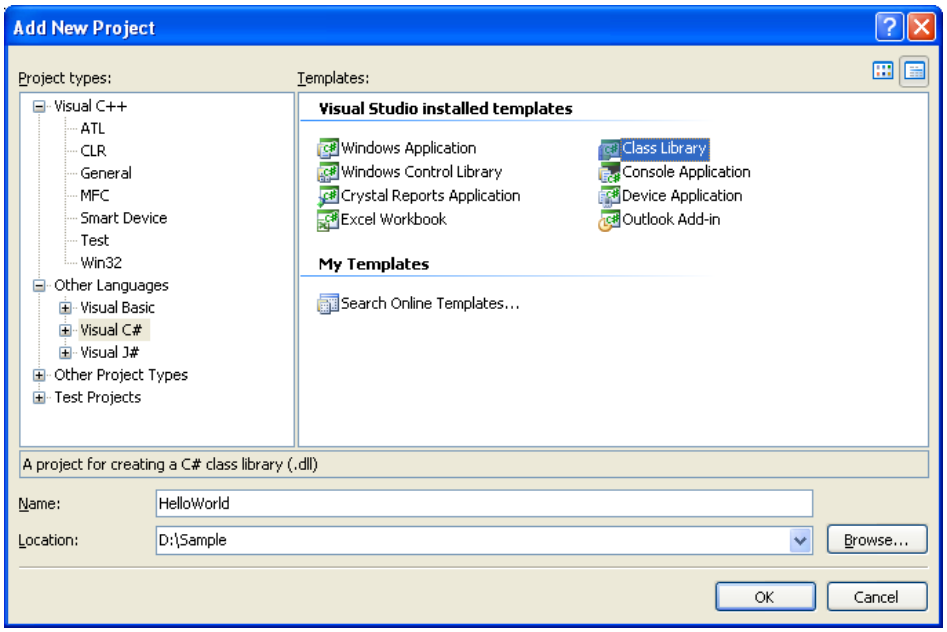



**Figure 2: Add New Project**

### 2.2.2 Add References

1. To add the RevitAPI reference:
   - From the View menu select Solution Explorer if the Solution Explorer window is not open.
   - In the Solution Explorer, right-click References to display a context menu.
   - From the context menu, click Add Reference. The Add Reference dialog box appears.
   - In the Add Reference dialog box, click the Browse tab. Locate the folder where Revit is installed and click the RevitAPI.dll. For example, the installed folder location is usually C:\Program Files\Revit Architecture 2008\Program\RevitAPI.dll.
   - Click OK to select the .dll and close the dialog box. RevitAPI appears in the Solution Explorer reference tree.
2. Add the System.Windows.Forms reference:
   - In the Solution Explorer, right-click References to display a context menu.
   - From the context menu, click Add Reference. The Add Reference dialog box appears.
   - In the Add Reference dialog box, click the .NET Tab.
   - From the Component Name list, select System.Windows.Forms.
   - Click OK to close the dialog box. System.Windows.Forms appears in the Solution Explorer reference tree.

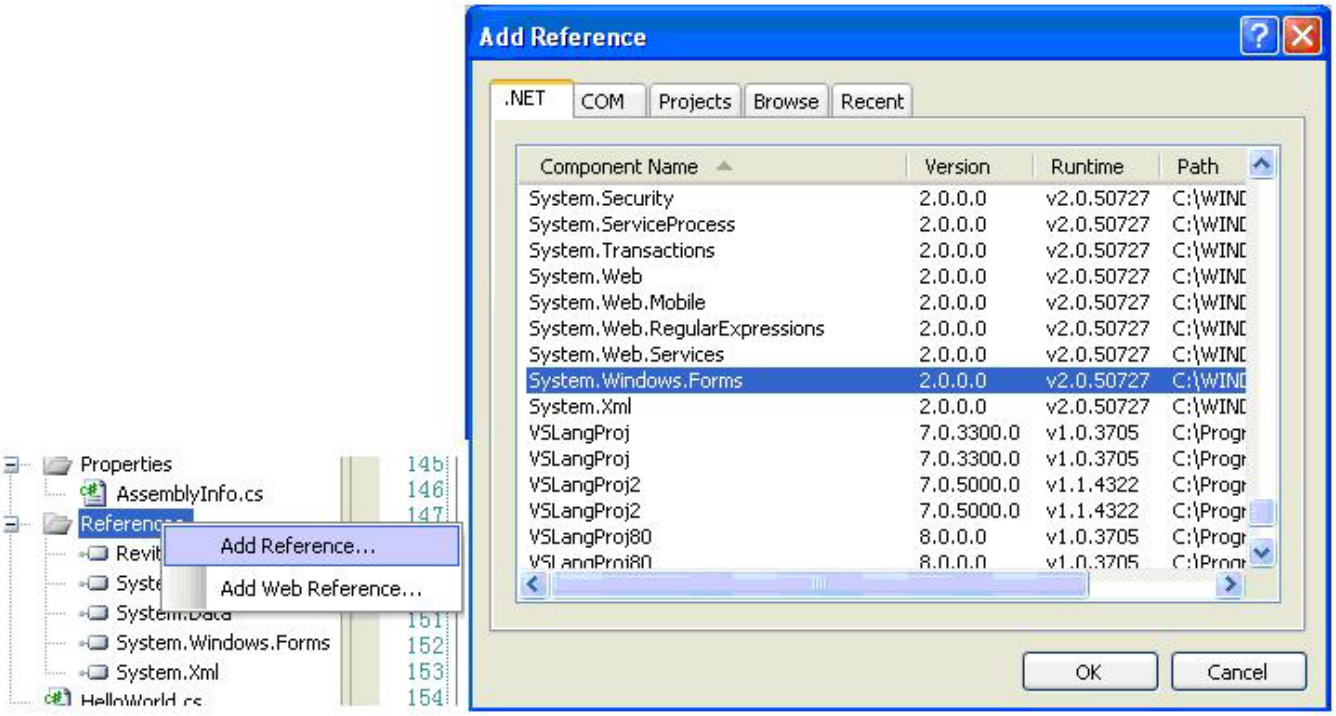



**Figure 3: Add Reference**

3. Set Copy Local:
   - In the Reference folder, right-click the RevitAPI node to display a context menu.
   - From the context menu, click Properties. The Properties dialog box appears.
   - Set Copy Local property to False.

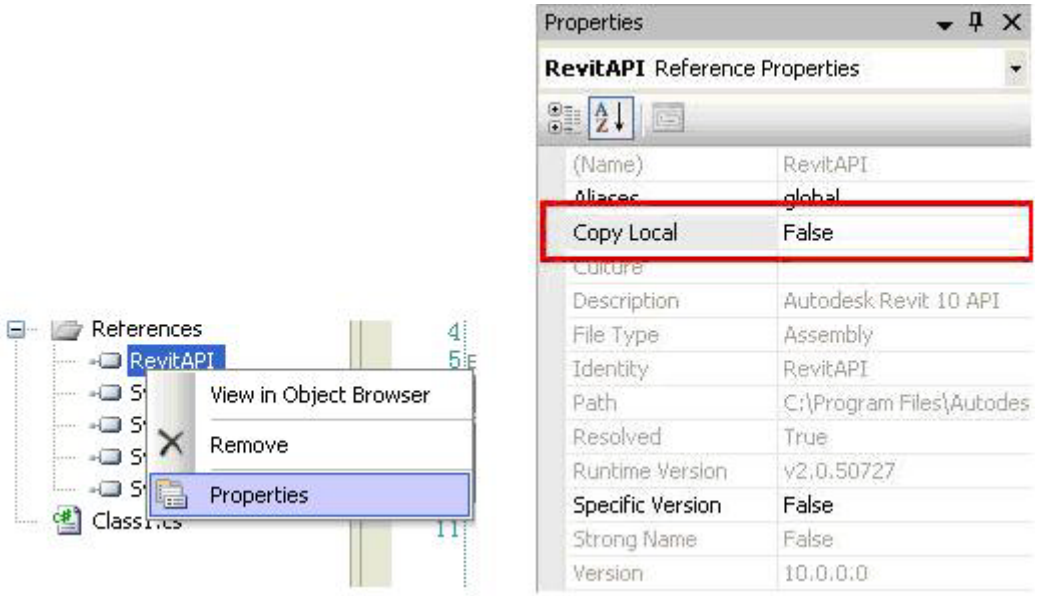



**Figure 4: Set Copy Local Property**

**Note:** Setting **Copy Local** to false tells Visual Studio not to copy the RevitAPI.dll file to the project output directory. There are usually two benefits to using this configuration.

- Saves space on your hard disk.

- Implements the add-in application fluently without a version conflict between the RevitAPI.dll and the Revit.exe.

### 2.2.3 Change the Class Name

To change the class name, complete the following steps:

1. In the class view window, right-click Class1 to display a context menu.
2. From the context menu, select Rename and change the class' name to CsHelloWorld.
3. In the Solution Explorer, right-click the Class1.cs file to display a context menu.
4. From the context menu, select Rename and change the file's name to CsHelloWorld.cs.
5. Double click CsHelloWorld.cs to open it for editing.

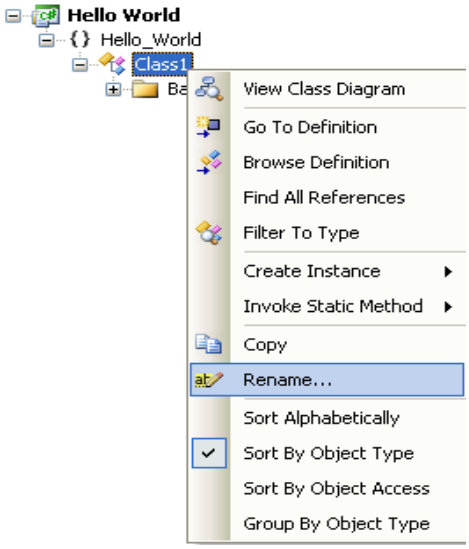



**Figure 5: Change the class name**

### 2.2.4 Add Code

Add the following code to create the add-in:

**Code Region 1- Getting Started**

```
using System;
using System.Windows.Forms;

using Autodesk.Revit;
namespace HelloWorld
{
   /// <summary>
   /// CsHelloWorld implements IExternalCommand interface
   /// </summary>
   public class CsHelloWorld:IExternalCommand
   {
      public IExternalCommand.Result Execute(ExternalCommandData revit,
       ref string message,ElementSet elements)
      {
```

```
            MessageBox.Show("Hello World");
            return IExternalCommand.Result.Succeeded;
        }
    }
}
```

**Note**: To use the MessageBox class, you must use the System.Windows.Forms namespace after you add the reference.

Every Revit add-in application must have an entry point class that implements the IExternalCommand interface, and you must implement the Execute method. The Execute method is the entry point for the add-in application similar to the Main method in other programs. The add-in entry point class definition is contained in an assembly. For more details, refer to the Add-in Integration chapter.

### 2.2.5 Build the Program

After completing the code, you must build the file. From the Build menu, click Build Solution. Output from the build appears in the Output window indicating that the project compiled without errors.

### 2.2.6 Modify the Revit.ini file

The HelloWorld.dll file appears in the project output directory. If you want to invoke the application in Revit, modify the Revit.ini file to register it into Revit.

1. To edit the Revit.ini file, open it for edit in Notepad. The Revit.ini file is usually located in the Revit installation directory on your computer. For example: C:\program Files\Revit Architecture 2008\Program.
2. Add the following to the end of the existing code:

```
[ExternalCommands]
ECCount=1
ECClassName1= HelloWorld.CsHelloWorld
ECAssembly1= D:\Sample\HelloWorld\bin\Debug\HelloWorld.dll
ECName1= HelloWorld
ECDescription1=Implementation of HelloWorld within Autodesk Revit
```

**Note**: ECAssembly1 is the path to the assembly, D:\Sample\HelloWorld\bin\Debug\HelloWorld.dll in this particular example.

Refer to the Add-in Integration chapter for more details about the Revit.ini file.

### 2.2.7 Debug the Add-in

Running a program in Debug mode uses breakpoints to pause the program so that you can examine the state of variables and objects. If there is an error, you can check the variables as the program runs to deduce why the value is not what you might expect.

1. In the Solution Explorer window, right-click the HelloWorld project to display a context menu.

2. From the context menu, click Properties. The Properties window appears.
3. Click the Debug tab.
4. Under the Start Action section, click Start external program and browse to the Revit.exe file. By default, the file is located at the following path, C:\Program File\Revit Architecture 2008\Program\Revit.exe.

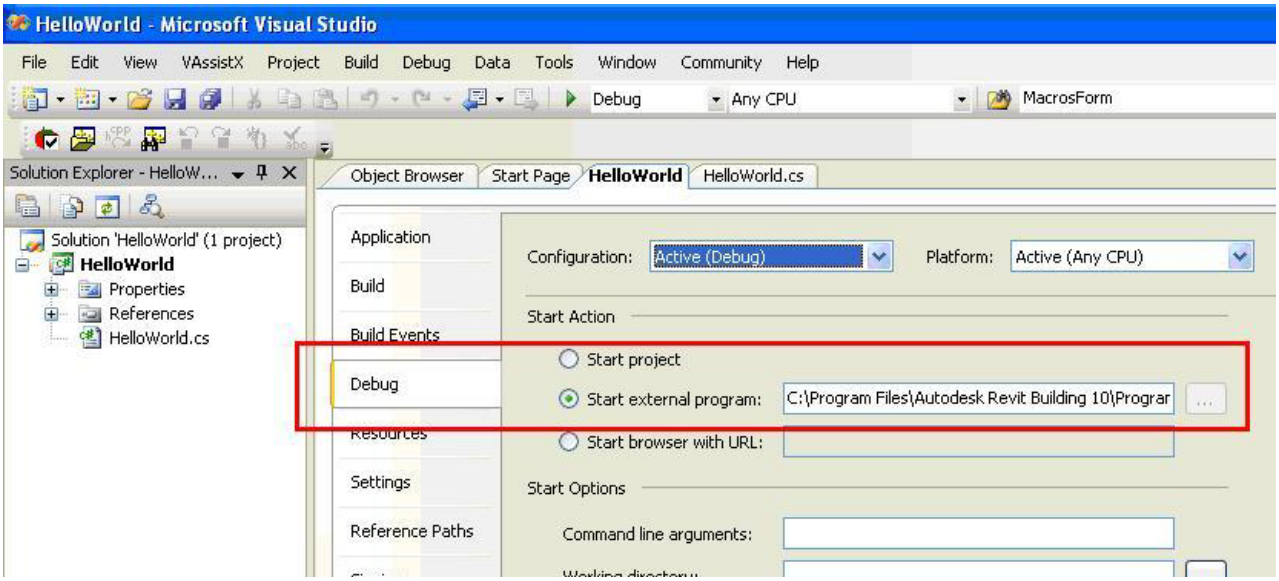



**Figure 6: Set debug environment**

5. From the Debug menu, select Toggle Breakpoint (or press F9) to set a breakpoint on the following line.

```
MessageBox.Show("Hello World");
```

6. Press F5 to start the debug procedure.
7. Test debugging:
   - From the Revit menu bar, click Tools > External Tools. HelloWorld appears in the submenu.

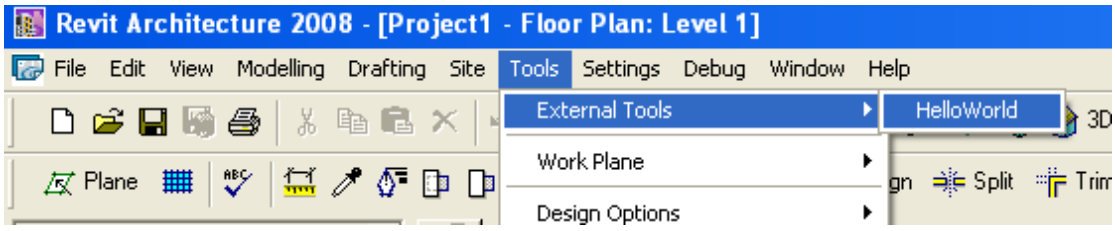



**Figure 7: HelloWorld External Tools command**

   - Click HelloWorld to execute the program, activating the breakpoint.
   - Press F5 to continue executing the program. The following system message appears.

**Figure 8: System message**

### 2.2.8 Troubleshooting

**Q:** Why does my add-in application not work?

**A:** There are usually three reasons the application does not work.

1. An error in Revit.ini configuration file.
2. An exception is thrown for a logical error in the code.
3. The .dll is compiled with a different RevitAPI version.

Begin by checking the Revit.ini file located in the Revit setup folder. Check the following eight possible causes:

1. Revit cannot find the Revit.ini file in the Revit setup folder.
2. Cannot find the [ExternalCommands] block in Revit.ini.
3. [ExternalCommands] is empty.
4. Location of [ExternalCommands] is incorrect.
5. ECCount is not equal to the real command number.
6. ECClassName is not correct.
7. [ExternalCommands] does not contain ECName.

Make sure the first six cases are corrected first. If they are not corrected, the add-in will not work. The last fix is not required. However, it is helpful to provide a name for your add-in since this name is used in the menu to identify your application. The default name for unnamed add-ins is usually External Tool.

After confirming the .ini file is configured correctly, check the add-in application code for an exception in the Execute method. There are usually two exceptions:

1. An exception occurs in Execute.
2. An exterior object definition or declaration.

For example:

```
Command: IExternalCommand
{
   A a = new A();//line x
   public IExternalCommand.Result Execute ()
   {
     //…
   }
}
Class A
```

```
{
  //…
}
```

The following two exceptions clearly identify the problem:

- An error in line x
- An exception is thrown in the Execute method.

**Note:** Use try..catch..finally in the example entry method to investigate the problem.

If the previous solutions do not resolve the problem and the error appears when you compile the sample code, the problem may be with the version of the RevitAPI used to compile the add-in. Delete the old RevitAPI reference and load a new one. For more details, refer to Add Reference.

## 2.3 Walkthrough: Add Hello World Menu

In the Walkthrough: Hello World section you learn how to create an add-in application and invoke it in Revit. You also learn to modify the Revit.ini file to register the add-in application as an external tool. Another way to invoke the add-in application in Revit is through a menu item.

### 2.3.1 Create a New Project

Complete the following steps to create a new project:

1. Create a C# project in Visual Studio using the Class Library template.
2. Type AddMenu as the project name.
3. Add the RevitAPI reference using the directions in the previous walkthrough, Walkthrough: Hello World.

### 2.3.2 Change the Class Name

To change the class name, complete the following steps:

1. In the class view window, right-click Class1 to display a context menu.
2. From the context menu, select Rename and change the class' name to CsAddMenu.
3. In the Solution Explorer, right-click the Class1.cs file to display a context menu.
4. From the context menu, select Rename and change the file's name to CsAddMenu.cs.
5. Double click CsAddMenu.cs to open it for editing.

### 2.3.3 Add Code

The Add Menu project is different from Hello World because it is automatically invoked when Revit runs. Use the IExternalApplication interface for this project. The IExternalApplication interface contains two abstract methods, OnStartup and OnShutdown. For more information about IExternalApplication, refer to the Add-in Integration chapter.

Add the following code for the menu item:

**Code region 2 - Getting Started**

```
public class CsAddMenu: IExternalApplication
{
   public IExternalApplication.Result OnShutdown(ControlledApplication application)
   {
```

```
      return IExternalApplication.Result.Succeeded;
   }

   public IExternalApplication.Result OnStartup(ControlledApplication application)
   {
      Autodesk.Revit.MenuItem menuItem = application.CreateTopMenu("NewMenuItem");
      //Create a menu iterm in the top menu “NewMenuItem”, click it,
     //the add-in application “HelloWorld” will be triggered.
      menuItem.Append(Autodesk.Revit.MenuItem.MenuType.BasicMenu, "HelloWorld",
       @"D:\HelloWorld.dll ", "HelloWorld.CsHelloWorld ");
      return IExternalApplication.Result.Succeeded;
   }
}
```

### 2.3.4 Build the Application

After completing the code, build the application. From the Build menu, click Build Solution. Output from the build appears in the Output window indicating that the project compiled without errors. AddMenu.dll is located in the project output directory.

### 2.3.5 Modify the Revit.ini File

To invoke the application in Revit, modify the Revit.ini file to register it into Revit.

1. Open the Revit.ini file for edit using Notepad. The.ini file is usually in the Revit installation directory on your computer. For example, it is usually located at the following path C:\Program Files\Revit Architecture 2008\Program.
2. Add the following information to the end of the file:

```
[ExternalApplications]
EACount = 1
EAClassName1 = AddMenu.CsAddMenu
EAAssembly1 = D:\Sample\AddMenu\AddMenu\bin\Debug\AddMenu.dll
```

**Note:** The AddMenu.dll file is in the default file folder in a new folder called Debug (D:\Sample\HelloWorld\bin\Debug\AddMenu.dll). Use the file path to evaluate ECAssembly1.

Refer to the Add-in Integration chapter for more information about the Revit.ini file.

### 2.3.6 Debugging

To begin debugging, build the project, and run Revit. A new menu item appears on the menu bar named NewMenuItem and HelloWorld appears in the drop-down menu.

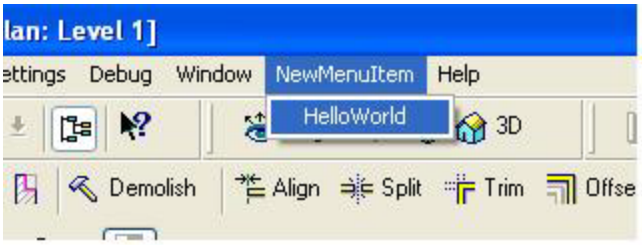



**Figure 9: Add a new menu Item to Revit**

Click HelloWorld to run the application and display the following dialog box.

**Figure 10: Hello World dialog box**

## 2.4 Walkthrough: Retrieve Selected Elements

Although the Hello World project is a very simple application, it gives you a good idea of how you can add new applications. This section introduces you to an add-in application that gets selected elements from Revit.

In the add-in applications, you perform a specific operation to a specific element. For example, you can get or change an element's parameter value. Complete the following steps to get a parameter value:

1. Create a new project and add the references using the previous walkthroughs.
2. Use the Application.ActiveDocument.Selection.Elements property to get the selected object.

The selected object is a Revit elementSet. You can get all of the information about the selected object from the elementSet. Use the IEnumerator interface or foreach loop to search the elementSet.

The following code is an example of how to retrieve selected elements.

**Code region 3 - Getting Started**

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms;

using Autodesk.Revit;
using Autodesk.Revit.Elements;

namespace Gettingstart
{
   public class GetSelectedElem : IExternalCommand
   {
      public IExternalCommand.Result Execute(Autodesk.Revit.ExternalCommandData commandData,
        ref string message, ElementSet elements)
```

```
      {
         // Before invoke this application, you can selete some elements
          //in Revit
         Document doc = commandData.Application.ActiveDocument;

         // Selection property returns an object that represents the active
          //selection
         Selection choices = doc.Selection;

         // The Elements property returns a set of the elements that were selected
         //at the time the external command was executed.
         ElementSet collection = choices.Elements;
         Element currentComponent = null;
         foreach (Element elem in collection)
         {
            currentComponent = elem;
            try
            {
               //…
            }
            catch (Exception e)
            {
               //…
            }
         }
         return IExternalCommand.Result.Succeeded;
      }
   }
}
```

After you get the selected elements, you can get the properties or parameters for the elements. For more information, see the Parameter chapter.

## 2.5 Walkthrough: Retrieve All Elements

Similar to retrieving selected elements, you can quickly get all project objects from the active document using the Application.ActiveDocument.Elements property. To query one or more elements, you can iterate through all elements and test each one.

**Code region 4 - Getting Started**

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms;

using Autodesk.Revit;
using Autodesk.Revit.Elements;
```

```
namespace Gettingstart
{
   public class GetAllElem : IExternalCommand
   {
      public IExternalCommand.Result Execute(Autodesk.Revit.ExternalCommandData commandData,
        ref string message, ElementSet elements)
      {
         // Get specified elements.
         Document doc = commandData.Application.ActiveDocument;
         ElementIterator i = doc.Elements;
         i.Reset();
         Wall wall = null;
         while (i.MoveNext())
         {
            Element e = i.Current as Element;
            if (null != e) // Get wall element.
            {
               wall = e as Wall;
               //…
            }
         }
         return IExternalCommand.Result.Succeeded;
      }
   }
}
```

# 3 Add-In Integration

Developers add functionality by creating and implementing External Commands and External Applications. Revit identifies the new commands and applications using the Revit.ini.

- External Commands appear in the Tools menu.
- External Applications are invoked when Revit starts up or shuts down

This chapter focuses on the following:

- Learning how to add functionality using External Commands and External Applications.
- How to access Revit events.
- How to customize, add, or edit menu items and toolbars.

## 3.1 Overview

The Revit API is set up based on Revit application functionality. Revit is not dependent on the API; RevitAPI is a class Library that only works when Revit is running. With the powerful API, you can add API based add-ins to extend Revit with the types and methods in the RevitAPI.

As the following picture shows, Revit Architecture, Revit Structure, and Revit MEP are specific to Architecture, Structure, and MEP respectively. In addition, Architecture also has other more widely used code, such as families.

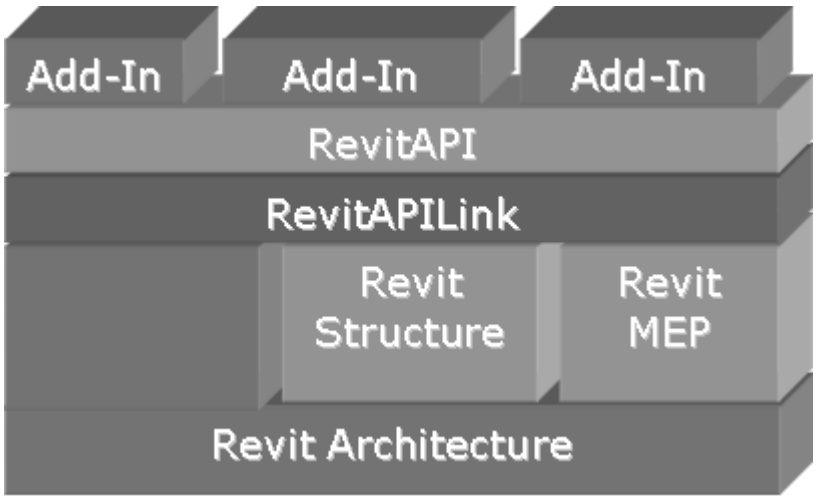



**Figure 11: Revit, RevitAPILink, RevitAPI and Add-in**

Specific entrypoint types are provided in the add-in .dll. These entrypoint classes implement the external-series interfaces, such as IExternalCommand and IExternalApplication. In this way, the add-in is invoked automatically on certain events or manually from the Tools menu.

IExternalCommand, IExternalApplication, and other available Revit events for add-in integration are introduced in this chapter.

## 3.2 External Command

External developers add functionality by implementing External Commands which appear in the Tools menu. Revit recognizes the existence of external commands using the Revit.ini

### 3.2.1 Mechanism

When no other commands or edit modes are active in Revit, the registered external command is enabled. When selected, the command object is created and the Execute method is called. Once this method returns back to Revit, the command object is destroyed. As a result, data cannot persist in the object between command executions. However, you can use the Revit shared parameters mechanism to store data in the Revit project.

You can integrate External Commands into Revit. There are two ways to trigger the commands:

- External Tools

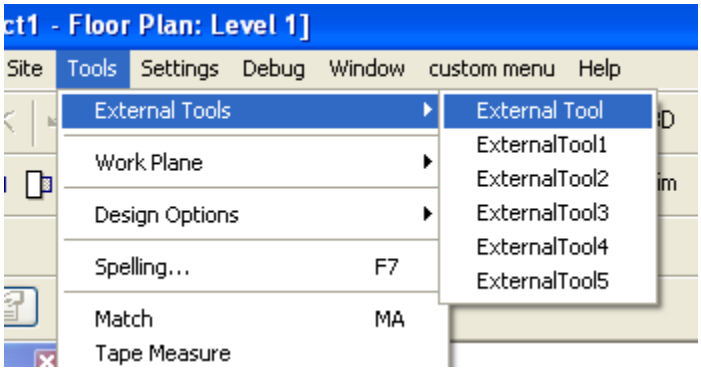



**Figure 12: External tools**

- Menu item

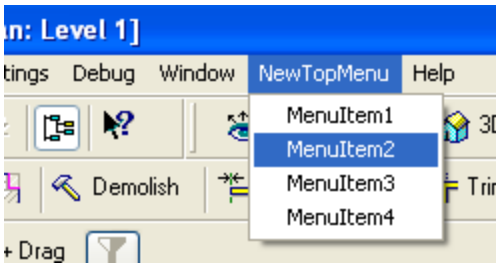



**Figure 13: MenuItems attached to the NewTopMenu**

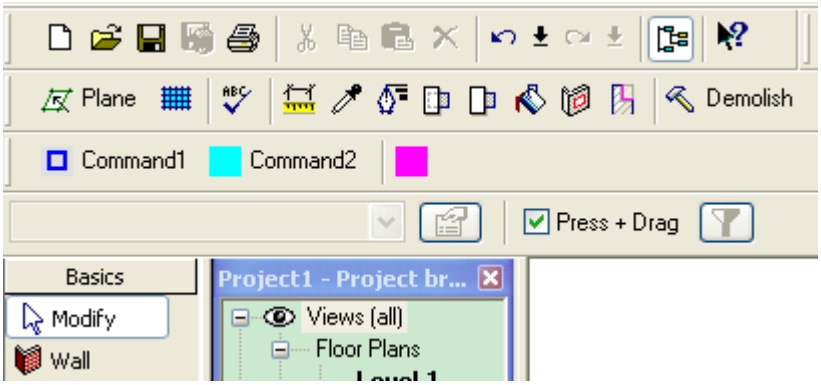



**Figure 14: MenuItems attached to the new toolbar**

The following diagram illustrates how the External Command is invoked

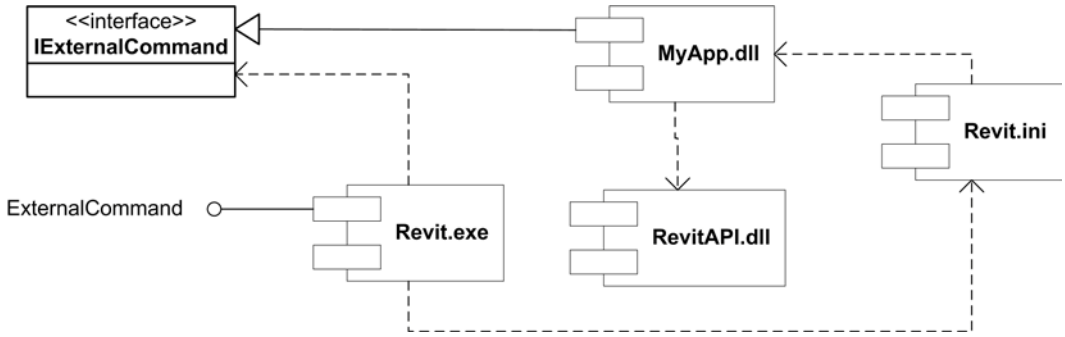



**Figure 15: Invoking the External Command**

You must initialize external tools and menu items upon start up. The initialization steps are as follows:

- Revit.ini is read. It identifies:
  - External Applications that can be invoked.
  - External Tools that can be added to the Revit Tools menu.
- External Application session adds menu items.

### 3.2.2 IExternalCommand

IExternalCommand is an interface implemented by the entrypoint type when you want to extend Revit with external-command add-ins. To add an external command, implement an object that supports the IExternalCommand interface.

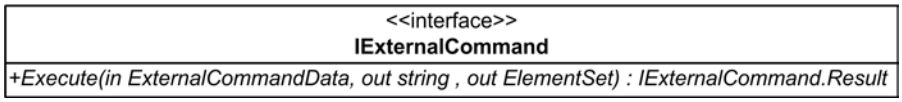



**Figure 16: IExternalCommand UML diagram**

The IExternalCommand interface has one abstract method, Execute, which is the main method for external commands. Overload the Execute method to implement IExternalCommand. The following code is the full Execute method definition:

```
public interface IExternalCommand
{
   IExternalCommand.Result Execute(Autodesk.Revit.ExternalCommandData commandData,
    ref string message, Autodesk.Revit.ElementSet elements);
}
```

The Execute method has three parameters:

- [in] commandData (ExternalCommandData)
- [out] message (String)
- [out] elements (ElementSet)

#### 3.2.2.1 [in] commandData (ExternalCommandData)

The ExternalCommandData object contains references to Application and View which is required by the external command. All Revit data is retrieved directly or indirectly from this parameter in the external command.

For example, the following statement illustrates how to retrieve Autodesk.Revit.Document from the commandData parameter:

```
Document doc = commandData.Application.ActiveDocument;
```

The following table illustrates the ExternalCommandData public properties

| Property | Description |
|---|---|
| Application (Autodesk.Revit.Application) | Retrieves an object that represents the current Application for external command. |
| Data (Autodesk.Revit.Collections.StringStringMap) | A data map that can be used to read and write data to the Revit journal file. |
| IsReadOnly (bool) | Informs the developer of the read and write capabilities of the object. (Inherited from APIObject.) |
| View (Autodesk.Revit.Elements.View) | Retrieves an object that represents the View external commands work on. |

**Table 1: ExternalCommandData public properties**

#### 3.2.2.2 [out] message (String):

Error messages are returned by an external command using the out parameter message. The string-type parameter is set in the external command process. When IExternalCommand.Result.Failed or IExternalCommand.Result.Cancelled is returned, and the message parameter is set, an error message appears.

The following code sample illustrates how to use the parameter message.

**Code Region 1 - Add-In Integration**

```
public class Command : Autodesk.Revit.IExternalCommand
{
   public Autodesk.Revit.IExternalCommand.Result Execute(
    Autodesk.Revit.ExternalCommandData commandData, ref string message,
    Autodesk.Revit.ElementSet elements)
   {
      message = "message test";
      return Autodesk.Revit.IExternalCommand.Result.Failed;
   }
}
```

Implementing the previous external command causes the following dialog box to appear:

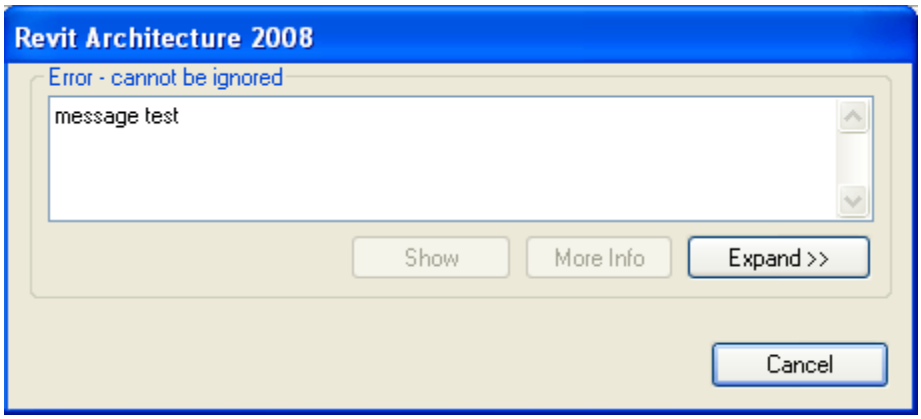



**Figure 17: Error message dialog box**

#### 3.2.2.3 [out] elements (ElementSet):

Whenever IExternalCommand.Result.Failed or IExternalCommand.Result.Canceled is returned and the parameter message is not empty, an error or warning dialog box appears. Additionally, if any elements are added to the elements parameter, these elements will be highlighted on screen. It is a good practice to set the message parameter whenever the command fails, whether or not elements are also returned.

The following code highlights pre-selected walls:

**Code Region 2 - Add-In Integration**

```
public Autodesk.Revit.IExternalCommand.Result Execute(
  Autodesk.Revit.ExternalCommandData commandData, ref string message,
  Autodesk.Revit.ElementSet elements)
{
   message = "Please take attention on the highlighted Walls!";
   Autodesk.Revit.ElementSet elems =
     commandData.Application.ActiveDocument.Selection.Elements;
   foreach (Autodesk.Revit.Element e in elems)
   {
      Autodesk.Revit.Elements.Wall wall = e as Autodesk.Revit.Elements.Wall;
      if (null != wall)
      {
         elements.Insert(e);
      }
   }
   return Autodesk.Revit.IExternalCommand.Result.Failed;
}
```

The following picture displays the result of the previous code.

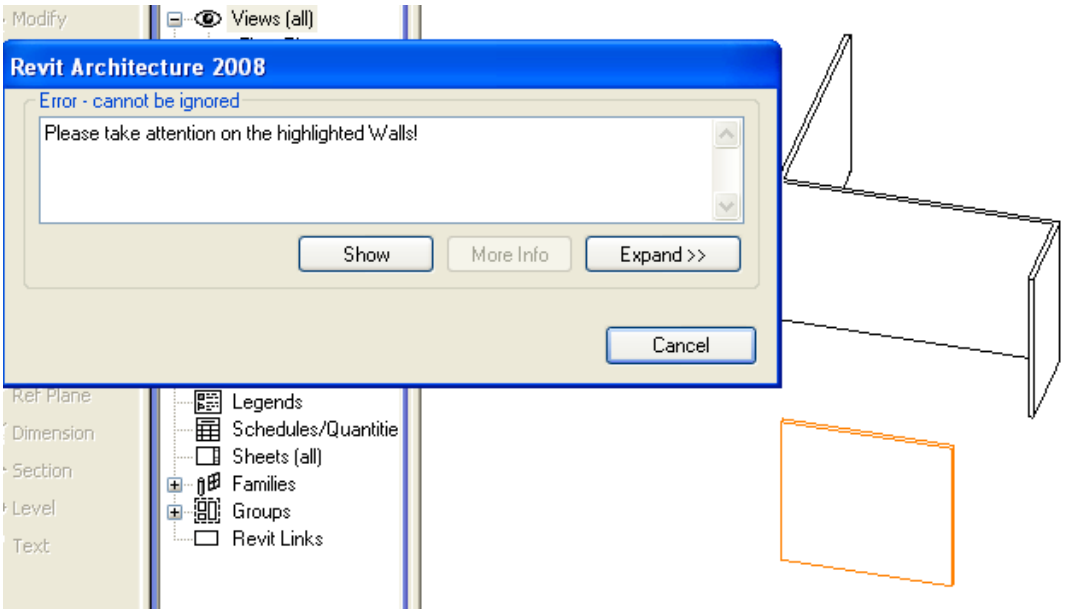



**Figure 18: Error message dialog box and highlighted elements**

#### 3.2.2.4 Return

The Return result indicates that the execution failed, succeeded, or is canceled by the user. If it does not succeed, Revit reverses changes made by the external command.

| Member Name | Description |
|---|---|
| IExternalCommand.Result.Succeeded | The external command completed successfully. Revit keeps all changes made by the external command. |
| IExternalCommand.Result.Failed | The external command failed to complete the task. Revit reverses operations performed by the external command, marking it a failure. |
| IExternalCommand.Result.Cancelled | The user cancelled the external command. Revit reverses changes made by the external command without marking it a failure. |

**Table 2: IExternalCommand.Result**

The following example displays a greeting message and allows the user to select the return value. Use the Execute method as the entrance to the Revit application.

**Code Region 3 - Add-In Integration**

```
public IExternalCommand.Result Execute(
  ExternalCommandData commandData, ref string message, ElementSet elements)
{
   try
   {
      Document doc = commandData.Application.ActiveDocument;
      ElementIdSet ids = doc.Delete(doc.Selection.Elements);

      DialogResult result = MessageBox.Show(
         "OK to return succeeded and delete all selection," +
         "No to cancel all commands.",
```

```
         "Revit", MessageBoxButtons.YesNoCancel);

      if (DialogResult.Yes == result)
      {
         return IExternalCommand.Result.Succeeded;
      }
      else if (DialogResult.No == result)
      {
         elements = doc.Selection.Elements;
         message = "Failed to delete selection.";
         return IExternalCommand.Result.Failed;
      }
      else
      {
         return IExternalCommand.Result.Cancelled;
      }
   }
   catch
   {
      message = "Unexpected Exception is thrown out.";
      return IExternalCommand.Result.Failed;
   }
}
```

### 3.2.3 [ExternalCommands]

In the Revit.ini [ExternalCommands] section, you can specify the external tools. External commands are listed in the [ExternalCommands] section. There can only be one [ExternalCommands] in the revit.ini file.

Example:

```
[ExternalCommands]
ECCount=1
ECClassName1=Project1.Class1
ECAssembly1=C:\Project1\Project1.dll
ECName1=”My Tool”
ECDescription1=”Implementation of My Tool within Revit”
```

**Note:** If the [ExternalCommands] section is not included in Revit.ini, you must create it.

- ECCount - Set the ECCount property to the total number of external commands available. For example, if your application supplies two external commands， the entry is:

```
[ExternalCommands]
ECCount=2
```

- ECClassName# - The ECClassName property is used to provide the command object that supports the IExternalCommand interface.
  - The command object is the full class name (with the namespace), such as MyNamespace.MyClass.
  - The # is the command number, such as ECClassName1.
- ECAssembly# - The ECAssembly property is used for .NET based objects only.
  - This property identifies the path to the Assembly containing the .NET command object.
  - The path can be relative to the Revit installation directory or fully qualified.
  - The # identifies the command number, such as ECAssembly1. The command number must match the command number used for the ECClassName property.

  In the previous example, the fully qualified path for the assembly is C:\Project1\Project1.dll.
- ECName# - The ECName property sets a short menu name for the external command in the Tools menu. The # identifies the command number, such as ECName1. The command number must match the command number used for the ECClassName property. In the previous example, the menu item for command #1 is “My Tool”.
- ECDescription# - The ECDescription property is a string that appears in the Revit status bar when the mouse moves over the menu item. The help string is a one-line description of the external command. In the previous example, the message “Implementation of My Tool within Revit” appears in the status bar when the mouse moves over the My Tool menu item.

**Note:** The Revit.ini file is read when Revit starts. It is not read at any other point during application execution.

## 3.3 External Application

External Developers can add functionality through External Applications as well as External Commands. Revit recognizes external applications using the Revit.ini. Menu items and the toolbar are customized using the External Application. Menu items and toolbar items are bound to an External command.

### 3.3.1 IExternalApplication

IExternalApplication is an interface implemented using the entrypoint type when you want to extend Revit functionality with an External Application. To add an External Application to Revit, you must implement an object that supports the IExternalApplication interface.

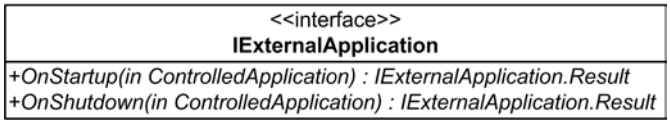



**Figure 19: IExternalCommand**

The IExternalApplication interface has two abstract methods, OnStartup and OnShutdown. Override the OnStartup and OnShutdown methods in the class implementing the interface to customize features before starting and after closing Revit.

Overload OnStartup and OnShutdown to implement IExternalApplication. The following code is the OnStartup and OnShutdown abstract definition:

```
public interface IExternalApplication
{
   IExternalApplication.Result OnShutdown(ControlledApplication application);
```

```
   IExternalApplication.Result OnStartup(ControlledApplication application);
}
```

The parameter application is the ControlledApplication type. ControlledApplication provides solutions to certain Revit events and customizes menu items and the toolbar. For example, the public event OnDialogBox of ControlledApplication can be used to capture the event of a dialog showing. The following code snippet registers the handling function that will be called right before a dialog is shown.

```
application.OnDialogBox += new Autodesk.Revit.Events.DialogBoxEventHandler(
  application_OnDialogBox);
```

The following code sample illustrates how to use the ControlledApplication type.

**Code Region 4 - Add-In Integration**

```
public class EApplication : Autodesk.Revit.IExternalApplication
{
   public Autodesk.Revit.IExternalApplication.Result OnShutdown(
    Autodesk.Revit.ControlledApplication application)
   {
      application.OnDialogBox -=
       new Autodesk.Revit.Events.DialogBoxEventHandler(application_OnDialogBox);
      application.OnDocumentNewed -=
       new Autodesk.Revit.Events.DocumentNewedEventHandler(application_OnDocumentNewed);
      return Autodesk.Revit.IExternalApplication.Result.Succeeded;
   }

   public Autodesk.Revit.IExternalApplication.Result OnStartup(
    Autodesk.Revit.ControlledApplication application)
   {
      application.OnDialogBox +=
       new Autodesk.Revit.Events.DialogBoxEventHandler(application_OnDialogBox);
      application.OnDocumentNewed +=
       new Autodesk.Revit.Events.DocumentNewedEventHandler(application_OnDocumentNewed);
      return Autodesk.Revit.IExternalApplication.Result.Succeeded;
   }

   void application_OnDialogBox(Autodesk.Revit.Events.DialogBoxData dialogBoxData)
   {
      MessageBox.Show("Before Dialog is shown!");
   }

   void application_OnDocumentNewed(Autodesk.Revit.Document document)
   {
      MessageBox.Show("Before create a new document!");
```

```
   }
}
```

### 3.3.2 [ExternalApplications]

In the Revit.ini, you specify external applications using [ExternalApplications]. External applications are listed in the [ExternalApplications] section. There is only one [ExternalApplications] section in the .ini file.

The following code example illustrates how to specify an external application:

```
[ExternalApplications]
EACount=1
EAClassName1=EA1.Class1
EAAssembly1=D:\EA1\bin\Debug\EA1.dll
```

**Note:** If the [ExternalApplications] section is not included in Revit.ini, you must create it.

- EACount - Set the EACount property to the total number of external applications available. For example, if your application supplies two external applications then the entry is:

```
[ExternalApplications]
EACount=2
```

- EAClassName# - The EAClassName property is used to provide the application object name that supports the IExternalApplication interface described earlier in this chapter.
  - For .NET objects it is the full class name (with the namespace), such as MyNamespace.MyClass.
  - The # identifies the application number such as EAClassName1.
- EAAssembly - The EAAssembly property is used for .NET based objects only.
  - This property identifies the Assembly path containing the .NET application object.
  - The path can be relative to the Revit installation directory or fully qualified.
  - The # identifies the application number such as EAAssembly1. The application number must match the application number used for the EAClassName property.

  In the previous example, the Assembly has the fully qualified path D:\EA1\bin\Debug\EA1.dll.

## 3.4 Access to Revit Events

Currently, you can access the following Revit Events:

- OnDialogBox
- OnDocumentClosed
- OnDocumentNewed
- OnDocumentOpened
- OnDocumentSaved
- OnDocumentSavedAs.

The following code provides more detail about how to use the Events:

```
namespace Autodesk.Revit
{
   public class Application : APIObject
   {
      public event DialogBoxEventHandler OnDialogBox;
      public event DocumentClosedEventHandler OnDocumentClosed;
      public event DocumentNewedEventHandler OnDocumentNewed;
      public event DocumentOpenedEventHandler OnDocumentOpened;
      public event DocumentSavedEventHandler OnDocumentSaved;
      public event DocumentSavedAsEventHandler OnDocumentSavedAs;
   }

   public class ControlledApplication
   {
      public event DialogBoxEventHandler OnDialogBox;
      public event DocumentClosedEventHandler OnDocumentClosed;
      public event DocumentNewedEventHandler OnDocumentNewed;
      public event DocumentOpenedEventHandler OnDocumentOpened;
      public event DocumentSavedEventHandler OnDocumentSaved;
      public event DocumentSavedAsEventHandler OnDocumentSavedAs;
   }
}
```

### 3.4.1 OnDialogBox

The OnDialogBox event is invoked before a form or a dialog box appears. The DialogBoxData parameter contains the necessary dialog data.

```
public delegate void DialogBoxEventHandler(DialogBoxData dialogBoxData);
```

### 3.4.2 OnDocumentClosed

The OnDocumentClosed event is invoked before a document is closed. The document parameter identifies the document to close.

```
public delegate void DocumentClosedEventHandler(Document document);
```

### 3.4.3 OnDocumentNewed

The OnDocumentNewed event is invoked before a document is created. The document parameter identifies the new document.

```
public delegate void DocumentNewedEventHandler(Document document);
```

### 3.4.4 OnDocumentOpened

The OnDocumentOpened event is invoked before a document is opened. The document parameter identifies the document to open.

```
public delegate void DocumentOpenedEventHandler(Document document);
```

### 3.4.5 OnDocumentSaved

The OnDocumentSaved event is invoked before a document is saved. The document parameter identifies the saved document.

```
public delegate void DocumentSavedEventHandler(Document document);
```

### 3.4.6 OnDocumentSavedAs

The OnDocumentSavedAs event is invoked before a document is saved as another file. The document parameter identifies the document that is saved as.

```
public delegate void DocumentSavedAsEventHandler(Document document);
```

## 3.5 Menu Item and Toolbar

Revit provides API solutions to integrate custom menus and toolbars. These APIs are used with IExternalApplication.

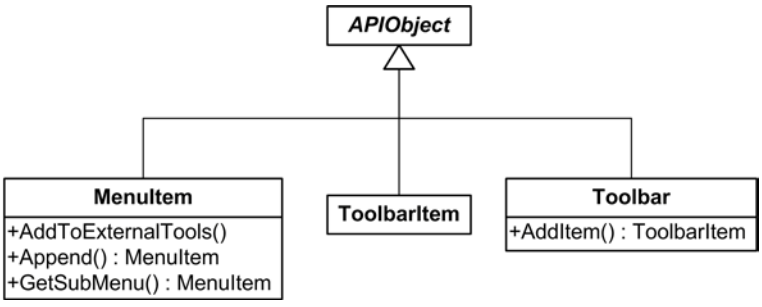



**Figure 20: Menu and Toolbar diagram**

### 3.5.1 Create a New TopMenu and Menu Items

The following sample illustrates how you can create menus and menu items in different ways. In this sample, BasicMenu, SeparatorMenu, and PopupMenu are added.

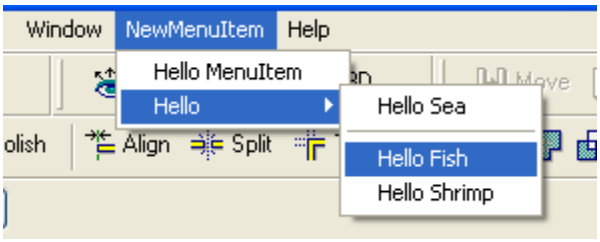



**Figure 21: New top menu and sub menu item**

**Figure 22: New top menu and sub menu item**

The following code integrates the menu and menu items in the previous figures. The external command location is D:\Hello.dll in an assembly containing the External Command Types:

- Hello.HelloMenuItem
- Hello.HelloSea
- Hello.HelloFish
- Hello.HelloShrimp.

**Code Region 5 - Add-In Integration**

```
//In the using segment add: using MenuItem = Autodesk.Revit.MenuItem
public Autodesk.Revit.IExternalApplication.Result OnStartup(
  Autodesk.Revit.ControlledApplication application)
{
   //Create a top menu
   MenuItem topMenuItem = application.CreateTopMenu("NewMenuItem");

   //Create a basic menu under the new top menu
   topMenuItem.Append(MenuItem.MenuType.BasicMenu, "Hello MenuItem",
     @"D:\Hello.dll", "Hello.HelloMenuItem");

   //Create a popup menu under the new top menu
   MenuItem popupMenu = topMenuItem.Append(MenuItem.MenuType.PopupMenu, "Hello");

   //Create three basic menus and a separator menu under the popup menu
   popupMenu.Append(MenuItem.MenuType.BasicMenu, "Hello Sea",
     @"D:\Hello.dll", "Hello.HelloSea");
   popupMenu.Append(MenuItem.MenuType.SeparatorMenu);
   popupMenu.Append(MenuItem.MenuType.BasicMenu, "Hello Fish",
     @"D:\Hello.dll", "Hello.HelloFish");
   popupMenu.Append(MenuItem.MenuType.BasicMenu, "Hello Shrimp",
     @"D:\Hello.dll", "Hello.HelloShrimp");
```

```
   return IExternalApplication.Result.Succeeded;
}
```

### 3.5.2 Create a New Toolbar

In the following example, create a new toolbar with several buttons. A dialog box appears when you click the Command1 button:

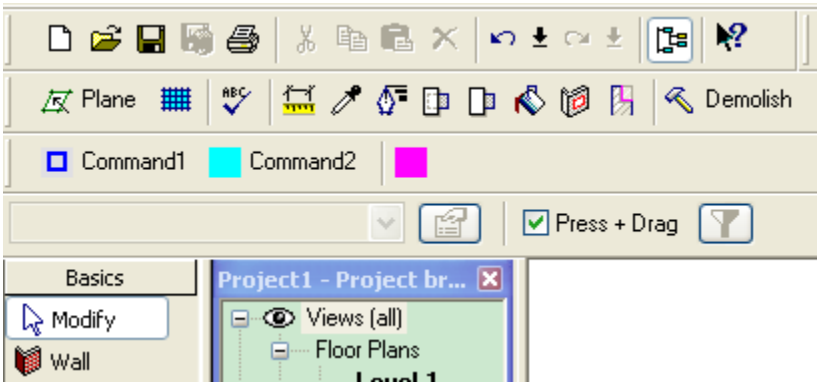



**Figure 23: New toolbar with three buttons**

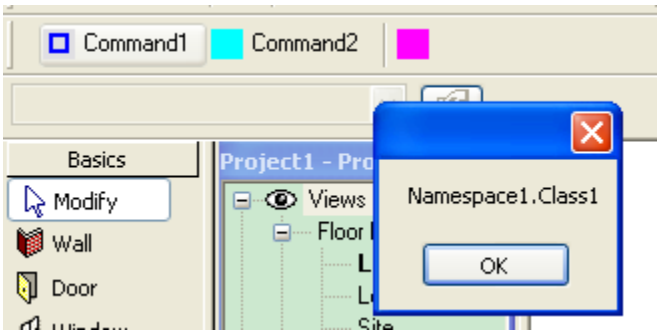



**Figure 24: Dialog box triggered by the Command1 button**

This following method creates a toolbar and several toolbar items in different ways. The imagePath parameter is the path to a toolbar bitmap graph, and the dllPath parameter is an assembly location with External Commands.

**Code Region 6 - Add-In Integration**

```
private void CreateToobarAndToolbarItems(
   ControlledApplication application, string imagePath, string dllPath)
{
   // Create a toolbar with named "Custom Toolbar" and set the image path.
   // Note: The image should be 16x*15 bmp(48*15, )
   Autodesk.Revit.Toolbar toolBar = application.CreateToolbar();
   toolBar.Name = "Custom Toolbar";
   toolBar.Image = imagePath;

   // Create one BtnRText toolbar item using menu item
   // Add a toolbar item using menu item,
   // the toolbar item calls the external command.
```

```
   MenuItem menuItem = application.CreateTopMenu("custom menu");
   MenuItem menuItem1 = menuItem.Append(MenuItem.MenuType.BasicMenu,
      "Command1", dllPath, "Namespace1.Class1");
   //Add the following menu item to create a new toolbar item.
   ToolbarItem item1 = toolBar.AddItem(menuItem1);
   //Tthe item is a button with rich text
   item1.ItemType = ToolbarItem.ToolbarItemType.BtnRText;
   item1.ItemText = "Command1";
   item1.StatusbarTip = "Command1 Tip";
   item1.ToolTip = "Command1 Tip";

   //Create the BtnRText toolbar item using the specified assembly
   //and class name. Add the second button with
   //Toolbar.AddItem(String, String).
   ToolbarItem item2 = toolBar.AddItem(dllPath, "Namespace2.Class2");
   //Button with rich text
   item2.ItemType = ToolbarItem.ToolbarItemType.BtnRText;
   item2.ItemText = "Command2";
   item2.StatusbarTip = "Command2 Tip";
   item2.ToolTip = "Command2 Tip";

   //Create a separator toolbar item and
   //add a separator between the second button and the third button
   ToolbarItem item3 = toolBar.AddItem("sep", "sep");
   //Separator button
   item3.ItemType = ToolbarItem.ToolbarItemType.BtnSeparator;

   //Create a standard toolbar item
   ToolbarItem item4 = toolBar.AddItem(dllPath, "Namespace3.Class3");
   //Standard item, image only.
   item4.ItemType = ToolbarItem.ToolbarItemType.BtnStd;
   item4.StatusbarTip = "Command3 Tip";
   item4.ToolTip = "Command3 Tip";
}
```

## 3.6 APIObject

Most reference types in the Revit API are inherited from the APIObject.

### 3.6.1 APIObject.IsReadOnly Property

The IsReadOnly property indicates whether the object is writable. If the property is set to true, an exception is raised when a method is called that requires write capabilities.

```
public virtual bool IsReadOnly { get; }
```

For example, Categories is set to IsReadOnly so that you cannot insert or remove an item. The Materials IsReadOnly setting is false so you can add and remove materials in the current document.

```
Application app = commandData.Application;
Document project = app.ActiveDocument;
Settings settings = project.Settings;
MessageBox.Show(settings.Categories.IsReadOnly.ToString());
MessageBox.Show(settings.Materials.IsReadOnly.ToString());
```

### 3.6.2 APIObject.ToString() Method

The method, APIObject.ToString(), returns the object type derived from the APIObject.

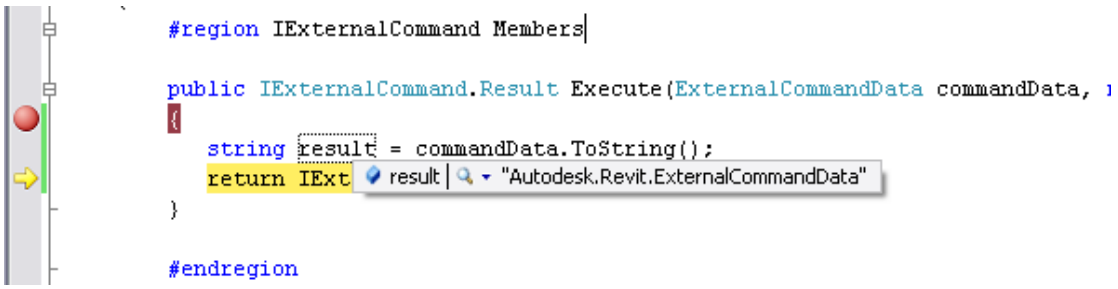



**Figure 25: APIObject.ToString ()**

The ToString method inherited from APIObject works the same way as the APIObject.ToString() method. However, you cannot use the ToString method to get object information such as the Element Name.

### 3.6.3 Equals

You cannot use the following to compare two elements.

- Object.Equals()
- == operator
- != operator

Compare Element and Category using their ElementId.Value. Some classes provide specific compare methods. For example, XYZ, UV, and Transform use AlmostEqual to compare two objects. If you want to compare specific elements, use a specific comparison method.

# 4 Application and Document

The Revit API uses a Multi-Document model similar to Microsoft Office. The top level classes in the Revit API are Application and Document.

- The Application object refers to an individual Revit session, providing access to documents, options, and other application-wide data and settings.
- The Document object is a single Revit project file representing a building model. Revit can have multiple projects open and multiple views for one project.
- The active, or top, view is the active project and the active document.

The relationship between the classes Application, Document, Element, and View are illustrated in the following diagram:

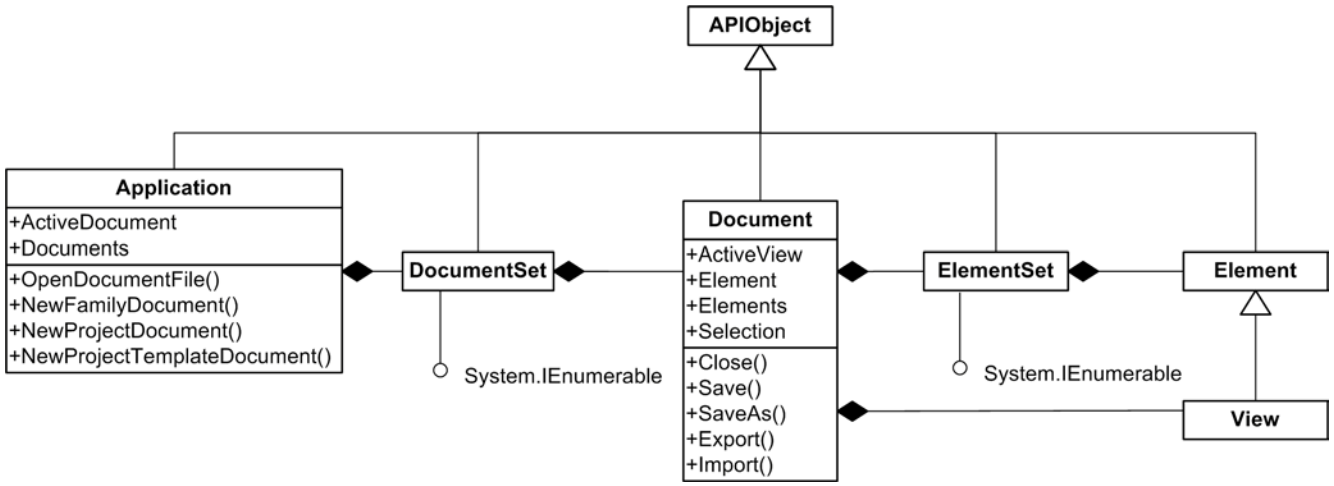



**Figure 26: Application-Document-View-Element diagram**

This chapter identifies all Application and Document functions, and then focuses on file management, settings, and units. For more details about the Element class, refer to the Elements Essentials and Editing Elements chapters and refer to the Views chapter for more details about the view elements.

## 4.1 Application Functions

Application Functions provide access to documents, objects, and other application data and settings. All application functions are identified and defined in the following sections.

### 4.1.1 Application Version Information

Application object properties include VersionBuild, VersionNumber and VersionName. Each property returns Revit version information.

### 4.1.2 Application-Wide Settings

The Options property provides configuration functions corresponding to the Setting > Options menu items. For more details, see the Settings section in this chapter.

### 4.1.3 Document Management

The Application class provides methods to open and create a document. For more details, see the Document and Management section in this chapter. In addition, the following characteristics apply:

- The active document is identified using the ActiveDocument property.

- Retrieve all open documents using the Documents property.

### 4.1.4 Shared Parameter Management

Revit uses one shared parameter file at a time. The Application.OpenSharedParameterFile method enables access to the shared parameter file whose path is set in the Options object. For more details, see the Shared Parameter chapter.

### 4.1.5 Menu and Toolbar Utility

Use the Application object to add new menus and custom toobar controls to Revit. You can add the following:

- Top level menus
- Pop-up menus
- Menu items
- Separators.

For more details, see the Menu Item and Toolbar section in the Add-In Integration chapter.

### 4.1.6 Event

The Revit API exposes document and dialog box events such as document open and save. Subscribing to these events notifies the application when the events are enabled and acts accordingly. For more details, see the Access to Revit Event section in the Add-In Integration chapter.

### 4.1.7 Create

The Create property returns an Object Factory used to create application-wide utility and geometric objects in the Revit API. Create is used when you create an object in the Revit application memory rather than your application's memory.

## 4.2 Document Functions

Document stores the Revit Elements, manages the data, and updates multiple data views. The Document class mainly provides the following functions.

### 4.2.1 Settings Property

The Settings property returns an object that provides access to general components within Revit projects. For more details, see the Settings section in this chapter.

### 4.2.2 Place and Locations

Each project has only one site location that identifies the physical project location on Earth. There can be several project locations for one project. Each location is an offset, or rotation, of the site location. For more details, see the Place and Locations chapter.

### 4.2.3 Type Collections

Document provides properties such as FloorTypes, WallTypes, and so on. All properties return a collection object containing the corresponding types loaded into the project.

### 4.2.4 View Management

A project document can have multiple views. The ActiveView property returns a View object representing the active view. You can traverse all elements in the project to retrieve other views. For more details, see the Views chapter.

### 4.2.5 Element Retrieval

The Document object stores elements in the project. Some properties enable you to retrieve elements from the Revit database.

- Retrieve all elements using the Elements property. This property provides access to all elements in the Document. It returns an iterator that traverses the entire element database and enables iteration for all elements.
- Retrieve selected elements using the Selection property. This property returns an object representing the active selection containing the selected project elements. It also provides an add-in UI interaction function Seletion.PickOne to pick elements.
- Retrieve certain elements by ElementId or UniqueId using the Element property.

For more details, see the Elements Essentials chapter.

### 4.2.6 File Management

Each Document object represents a Revit project file. Document provides the following functions:

- Retrieve file information such as file path name and project title.
- Provides Close and Save methods to close and save the document.

For more details, see the Document and File management section in this chapter.

### 4.2.7 Element Management

Revit maintains all Element objects in a project.

- The Create property returns an Object Factory used to create new project element instances in the Revit API, such as FamilyInstance or Group.
- Use the Delete method to delete an element in the project.
  - Deleted elements are not displayed in any views and are removed from the Document and any dependent elements.
  - References to deleted elements are invalid and cause an exception.

  For more details, see the Editing Element chapter.
- Document performs the following common operations on elements in the document such as
  - Move
  - Rotate
  - Mirror
  - Array.

  For more details, see the Editing Element chapter.

### 4.2.8 Transactions

Every external command is inside a transaction. A transaction combines several operations into one atomic operation. If an external command is successful, that is if it returns Result.Succeeded, the transaction will commit all the changes and close itself. The operation is completed. If, on the other hand, the external command is not successful, the transaction will abort, which in effect will cancel all operations within. For more details, see the Transaction chapter.

#### 4.2.8.1 Events

Events are raised when you save a project using Save or Save As. To capture the events and respond in the application, you must register the event handlers. For more details, see the Access to Revit Event section in the Add-In Integration chapter.

### 4.2.9 Others

Document also provides other functions:

- ParameterBindings Property - Mapping between parameter definitions and categories. For more details, see the Shared Parameter chapter.
- ReactionsAreUpToDate Property - Reports whether the reactionary loads changed. For more details, see the Loads section in the Revit Structure chapter.

## 4.3 Document and File Management

Document and file manangement make it easier to create and find your documents and files so that you can work in them. Using the Revit API, you can load family symbols and family instances.

### 4.3.1 Document Retrieval

The Application class maintains all documents. As previously mentioned, you can open more than one document in a session. When you open more than one document, the active, or the top view, identifies the document you are using. The active document is retrieved using the Application class property, ActiveDocument.

All open documents, including the active document, are retrieved using the Application class Documents property. The property returns a set containing all open documents in the Revit session.

### 4.3.2 Document File Information

The Document class provides two properties for each corresponding file, PathName, and Title.

- PathName returns the document's fully qualified file path. The PathName returns an empty string if the project has not been saved since it was created.
- Title is the project title, which is usually derived from the project filename. The returned value varies based on your system settings.

### 4.3.3 Open a Document

The Application class provides a method to open an existing project file:

| Method | Corresponding UI Command | Event |
|---|---|---|
| Document OpenDocumentFile(string filename ) | File>Open... | OnDocumentOpened |

**Table 3: Open Document in API**

When you specify a string with a fully qualified file path, Revit opens the file and creates a Document instance. Use this method to open a file on other computers by assigning the files Universal Naming Conversion (UNC) name to this method.

The file can be a project file with the extension .rvt, a family file with the extension .rfa, or a template file with the extension .rte. The method throws System.InvalidOperationException if you try to open unsupported file types. If the document is opened successfully, the OnDocumentOpened event is raised.

### 4.3.4 Create a Document

Create new documents using the Application methods in the following table.

| Method | Corresponding UI Command | Event |
|---|---|---|
| Document NewProjectDocument(<br>string templateFileName<br>); | File>New>Project... | OnDocumentNewed |
| Document NewFamilyDocument(<br>string templateFileName<br>); | File>New>Family...<br>File>New>TitleBlock...<br>File>New>Annotation Symbol... | OnDocumentNewed |
| Document NewProjectTemplateDocument(<br>string templateFilename<br>); | File>New>Project... | OnDocumentNewed |

**Table 4: Create Document in the API**

Each method requires a template file name as the parameter. The created document is returned based on the template file.

### 4.3.5 Save and Close a Document

The Document class provides methods to save or close instances.

| Method | Corresponding UI Command | Event |
|---|---|---|
| Save | File>Save | OnSave |
| SaveAs | File>Save As... | OnSaveAs |
| Close | File>Close | OnClose |

**Table 5: Save and Close Document in API**

**Note**: The Close method does not affect the active document or raise the OnClose event, because the document is used by an external application. You can only call this method on non-active documents. Enhancements are planned in future versions.

### 4.3.6 Load Family and Load Family Symbol

The Document class provides you with the ability to load an entire family and all of its types/symbols into the project. Because loading an entire family can take a long time and a lot of memory, the Document class provides a similar method, LoadFamilySymbol to load only specified symbols.

For more details, see the Component Elements chapter.

## 4.4 Settings

On the Settings menu, the commands in boxes in the figure below are available in the Revit API

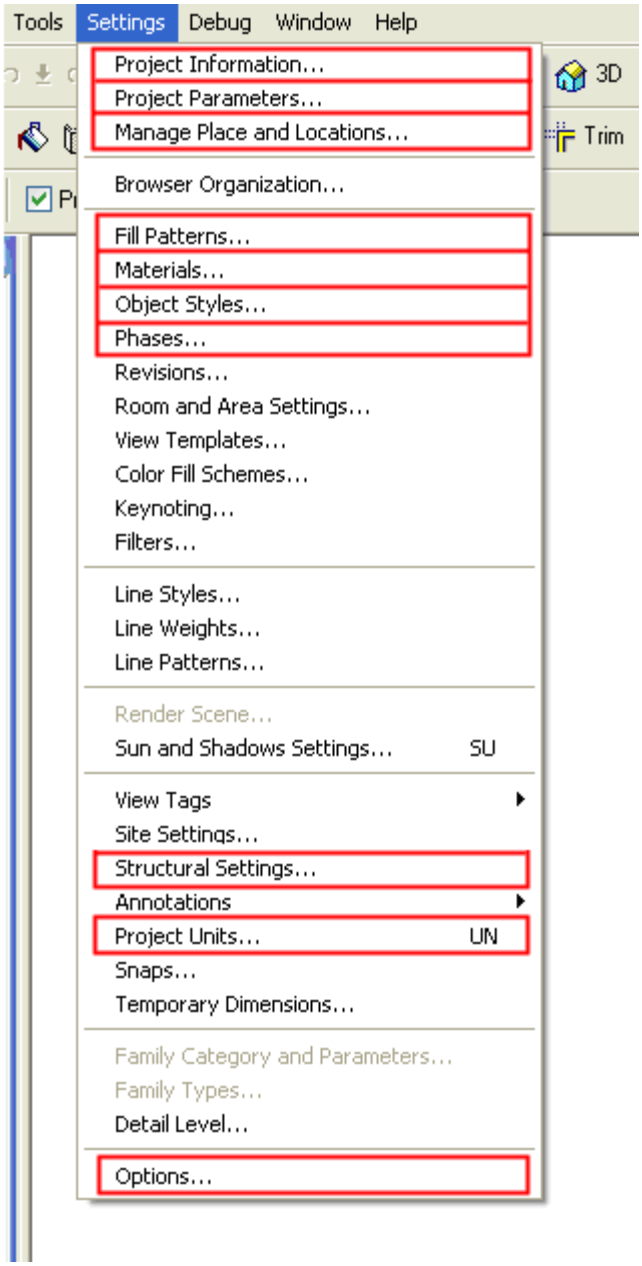



**Figure 27: Setting menu items**

The following table identifies the APIs and the corresponding UI commands.

| UI command | Associated API | Reference |
|---|---|---|
| Project Information... | Document.ProjectInformation | See the following note |
| Project Parameters... | Document.ParameterBindings (Only for Shared Parameter) | See Shared Parameter |
| Manage Place and Locations... | Document.ProjectLocations<br>Document.ActiveProjectLocation | See Place and Locations |
| Fill Patterns... | Document.Settings.FillPatterns | See the following note |
| Materials... | Document.Settings.Materials | See Materials Management |
| Object Styles... | Document.Settings.Categories | See the following note |

| UI command | Associated API | Reference |
|---|---|---|
| Phases... | Document.Phases | See the following note |
| Structural Settings... | Only exposed Loads related | See Revit Structure |
| Project Units... | Document.ProjectUnit | See Unit |
| Options... | Application.Options | See the following note |

**Table 6: Settings in API and UI**

**Note**:

For more information about some of the commands in the table, read the following information:

- Project Information - The API provides the ProjectInfo class which is retrieved using Document.ProjectInformation to represent project information in the Revit project.

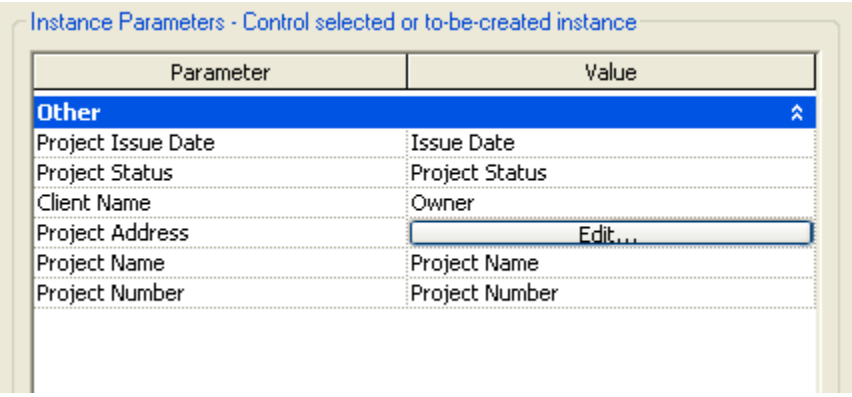



**Figure 28: Project information parameters**

The following table identifies the corresponding APIs for the Project Information parameters.

| Parameters | Corresponding API | Built-in Parameters |
|---|---|---|
| Project Issue Date | ProjectInfo.IssueDate | PROJECT_ISSUE_DATE |
| Project Status | ProjectInfo.Status | PROJECT_STATUS |
| Client Name | ProjectInfo.ClientName | CLIENT_NAME |
| Project Address | ProjectInfo.Address | PROJECT_ADDRESS |
| Project Name | ProjectInfo.Name | PROJECT_NAME |
| Project Number | ProjectInfo.Number | PROJECT_NUMBER |

**Table 7: ProjectInformation**

Use the properties exposed by ProjectInfo to retrieve and set all strings. These properties are implemented by the corresponding built-in parameters. You can get or set the values through built-in parameters directly. For more information about how to gain access to these parameters through the built-in parameters, see the Parameter section in the Elements Essentials chapter. However, the recommended way to get project information is to use the ProjectInfo properties.

- Fill Patterns - Retrieve all Fill Patterns in the current document using Document.Settings.FillPatterns.

- Object Styles - Use Settings.Categories to retrieve all information in Category objects except for Line Style. For more details, see the Categories section in the Elements Essentials chapter and Material chapter.
- Phases - Revit maintains the element lifetime by phases, which are distinct time periods in the project lifecycle. All phases in a document are retrieved using the Document.Phases property. The property returns an array containing Phase class instances. However, Revit does not expose functions from the Phase class.
- Options - The Options command configures project global settings. You can retrieve an Options.Application instance using the Application.Options property. Currently, the Options.Application class only supports access to library paths and shared parameters file.

## 4.5 Unit

The Revit Unit System is based on seven base units for seven base quantities that are independent. The base units are identified in the following table.

| Base Unit | Unit In Revit | Unit System |
|---|---|---|
| Length | Feet (ft) | Imperial |
| Angle | Radian | Metric |
| Mass | Kilogram (kg) | Metric |
| Time | Seconds (s) | Metric |
| Electric Current | Ampere (A) | Metric |
| Temperature | Kelvin (K) | Metric |
| Luminous Intensity | Candela (cd) | Metric |

**Table 8: 7 Base Units in Revit Unit System**

**Note**: Since Revit stores lengths in feet and other basic quantities in metric units, a derived unit involving length will be a non-standard unit based on both the Imperial and the Metric systems. For example, since a force is measured in “mass-length per time squared”, it is stored in kg-ft / $s^2$.

The Revit API provides the information in the following dialogs using Document.ProjectUnit.

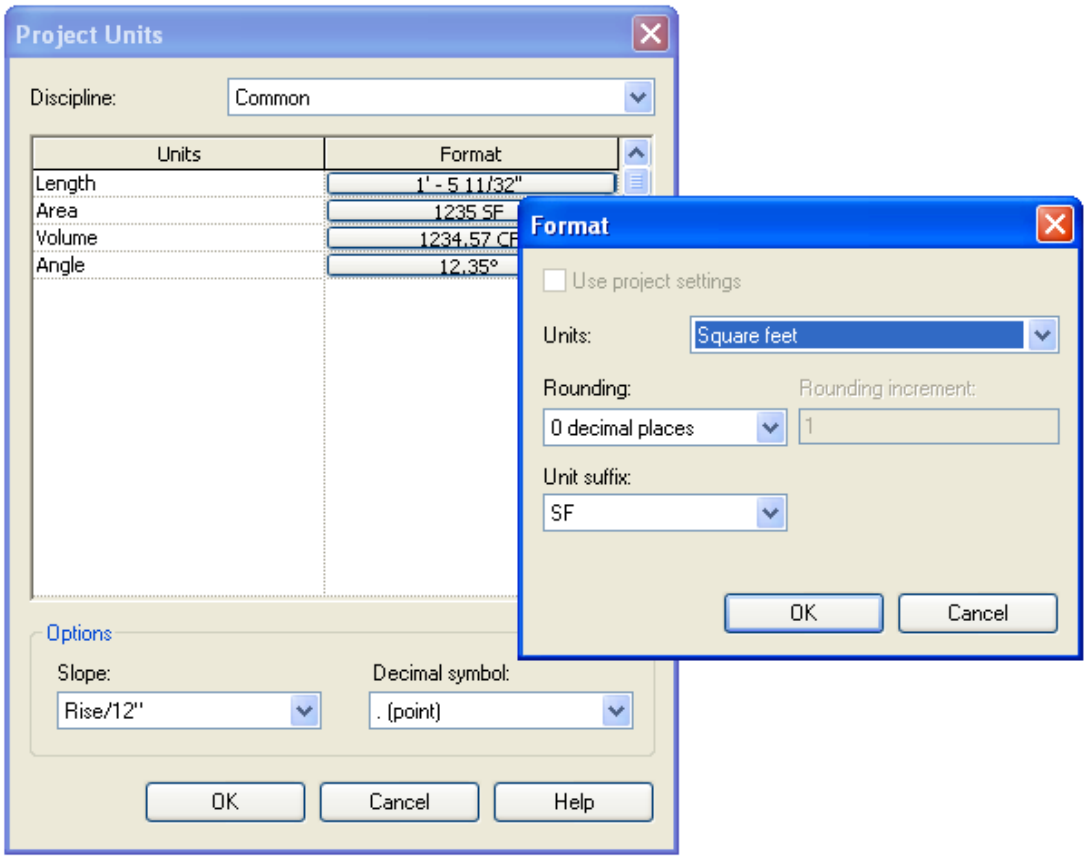



**Figure 29: Project units and format dialog box**

The APIs used to gain access to the settings include the following

| UI Command | Corresponding Type in API | Access API |
|---|---|---|
| Slope | Enums.RiseRunOrAngleType | ProjectUint.Slope |
| Decimal symbol | Enums.DecimalSymbolType | ProjectUint.DecimalSymbolType |
| Discipline | Enums.UnitDiscipline | ProjectUnit.FormatOptions.Discipline |
| Units | Enums.DisplayUnitType | ProjectUnit.FormatOptions.Units |
| Rounding | System.Double | ProjectUnit.FormatOptions.Rounding |
| Unit suffix | Enums.UnitSuffixType | ProjectUnit.FormatOptions.Unitsuffix |

**Table 9: Project Unit Properties**

# 5 Elements Essentials

An Element corresponds to a single building or drawing component, such as a door, a wall, or a dimension. In addition, an Element can be abstract, like a door type, a view, or a material definition. If a building is regarded as a system, the components are Elements.

This chapter introduces the Revit API Elements starting with an overview of the Revit Elements classification. Detailed Element classification is next followed by a discussion of important Element properties.

## 5.1 Elements Classification

The Revit API classifes Elements based on the Element specialty. Revit Elements are divided into six groups. Each group contains relative Elements and their corresponding symbols. The following digram identifies each Element group and the corresponding symbol.

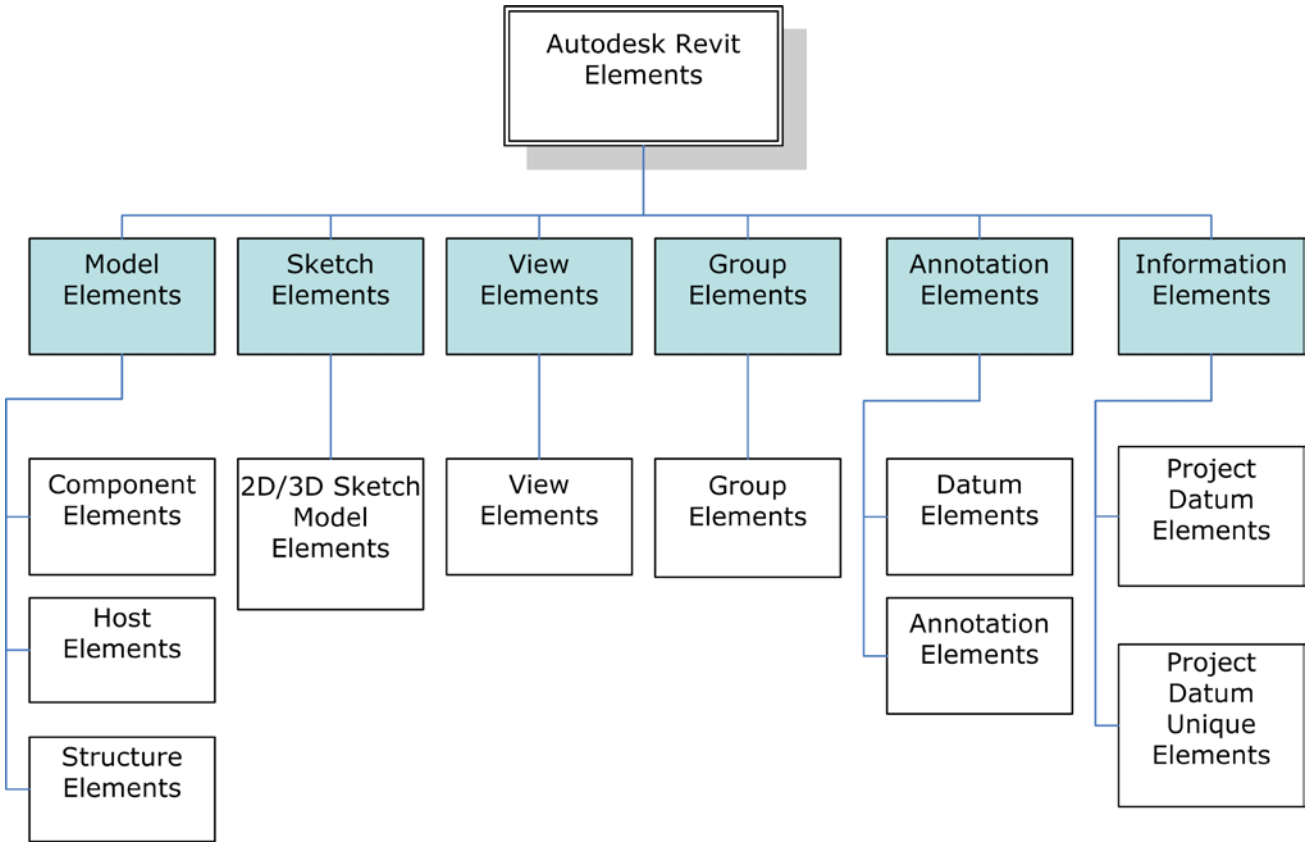



**Figure 30: Elements classification diagram**

### 5.1.1 Model Elements

Model Elements represent physical items that exist in one building project. Elements in the Model Elements group can be subdivided into the following:

- Component Elements - Component Elements contain component family objects. You can load family objects into your project or create them from family templates. For more information, see the Component Elements chapter.
- Host Elements - Host Elements contain system family objects that can contain other model elements, such as wall and floor. For more information about Host Elements, see the Host Elements and Opening chapter.

- Structure Elements. - Structure Elements contain elements that are only used in Revit Structure. For more information about Structure Elements, see the Revit Structure chapter.

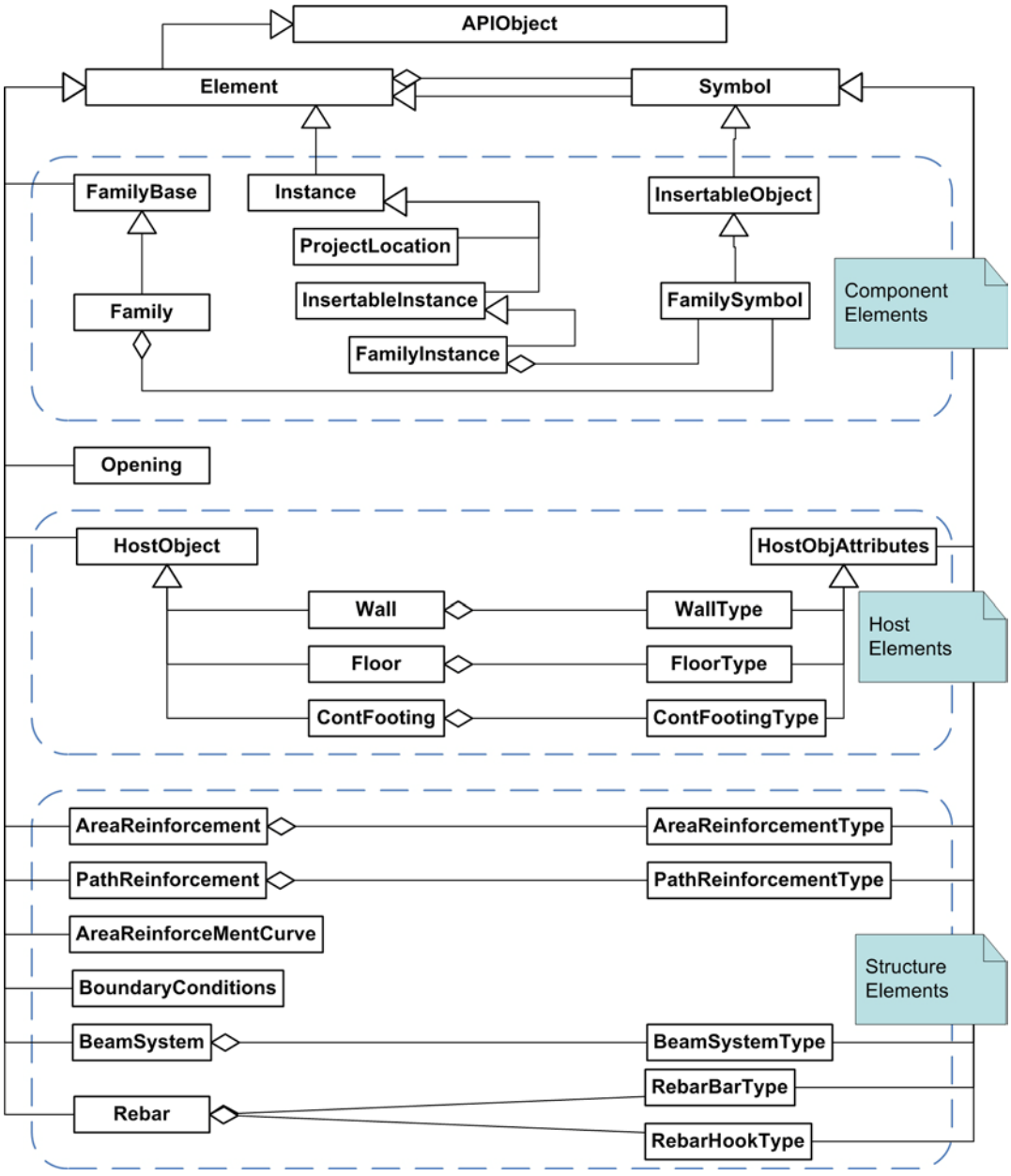



**Figure 31: Model Elements diagram**

### 5.1.2 View Elements

View Elements represent the way you view and interact with other objects in Revit. For more information, see the Views chapter.

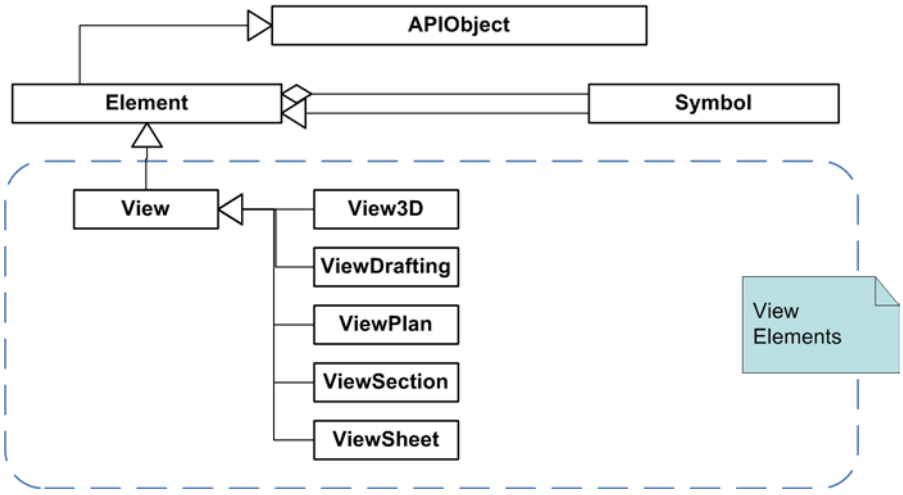



**Figure 32: View Elements diagram**

### 5.1.3 Group Elements

Group Elements represent the assistant Elements such as Array and Group objects in Revit. For more information, see the Editing Elements chapter.

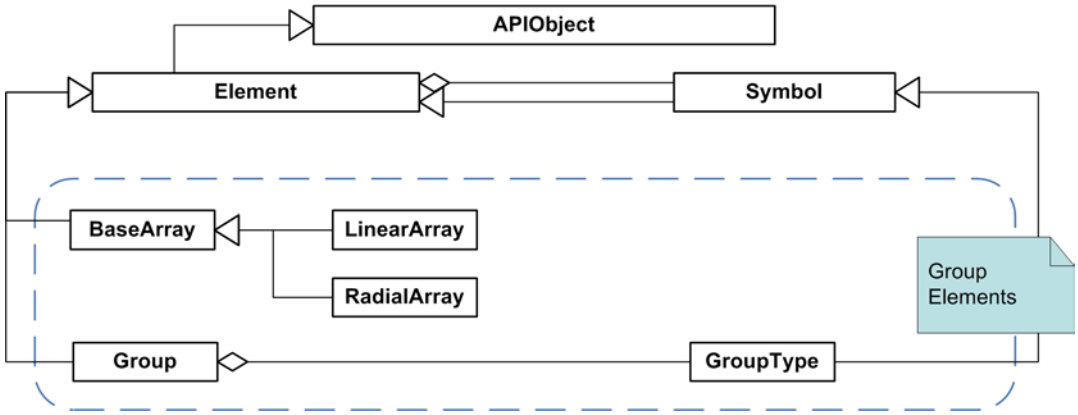



**Figure 33: Group Elements UML diagram**

### 5.1.4 Annotation and Datum Elements

Annotation and Datum Elements contain non-physical items that are visible.

- Annotation Elements represent 2D components that maintain scale on paper and are only visible in one view. For more information about Annotation Elements, see the Annotation Elements chapter. For Room information, see the Revit Architecture chapter.

**Note:** Annotation Elements representing 2D components do not exist only in 2D views. For example, dimensions can be drawn in 3D view while the shape only exists in a 2D planar face.

- Datum Elements represent non-physical items that are used to establish project context. These elements can exist in views. Datum Elements are further divided into the following:
  - Common Datum Elements - Common Datum Elements represent non-physical visible items used to store data for modeling.
  - Datum FamilyInstance - Datum FamilyInstance represents non-physical visible items loaded into your project or created from family templates.

**Note:** For more information about Common Datum Elements and Datum FamilyInstance, see the Datum and Information Elements chapter; for ModelCurve related contents, see the Sketching chapter.

- Structural Datum Elements - Structural Datum Elements represent non-physical visible items used to store data for structure modeling. For more information about Structural Datum Elements, see the Revit Structure chapter.

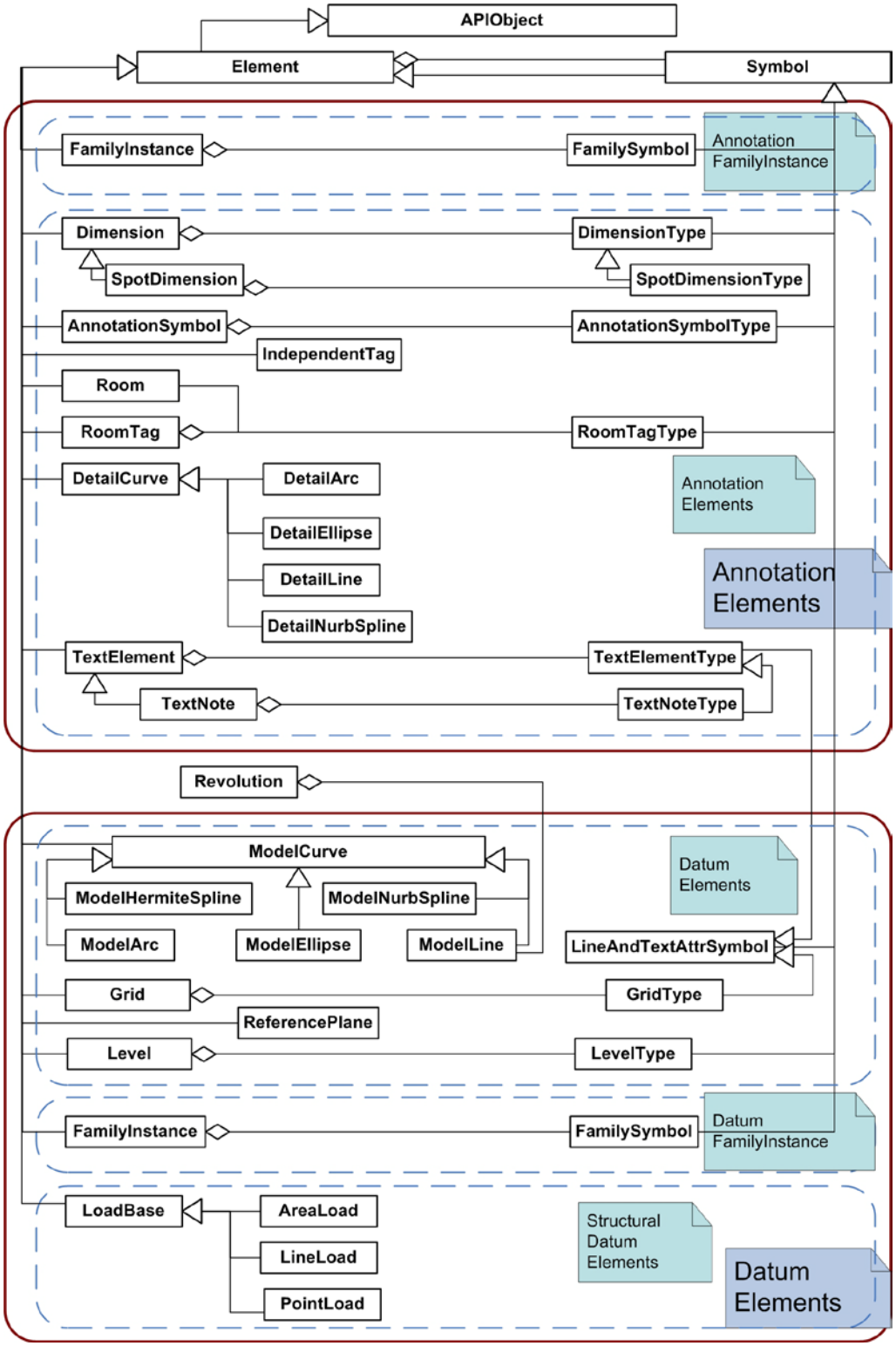



**Figure 34: Annotation Elements diagram**

### 5.1.5 Sketch Elements

Sketch Elements represent temporary items used to sketch 2D/3D form. This group contains the following objects used in family modeling and massing

- SketchPlane
- Sketch
- Path3D
- GenericForm.

For Sketch details, see the Sketching chapter.

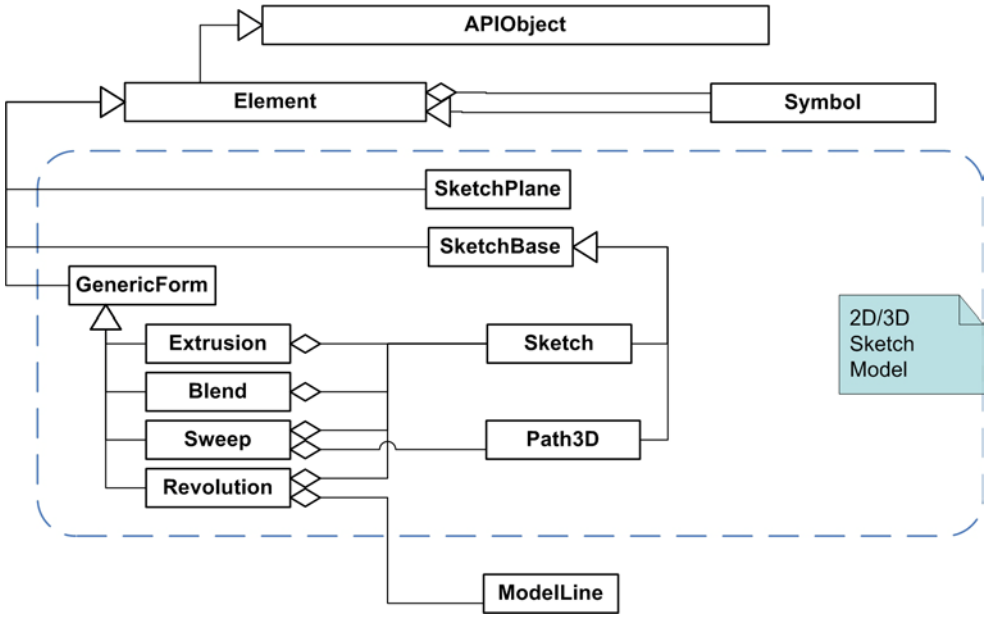



**Figure 35: Sketch Elements UML diagram**

### 5.1.6 Information Elements

Information Elements contain non-physical invisible items used to store project and application data. Information Elements are further separated into the following:

- Project Datum Elements
- Project Datum Elements (Unique).

For more information about Datum Elements, see the Datum and Information Elements chapter.

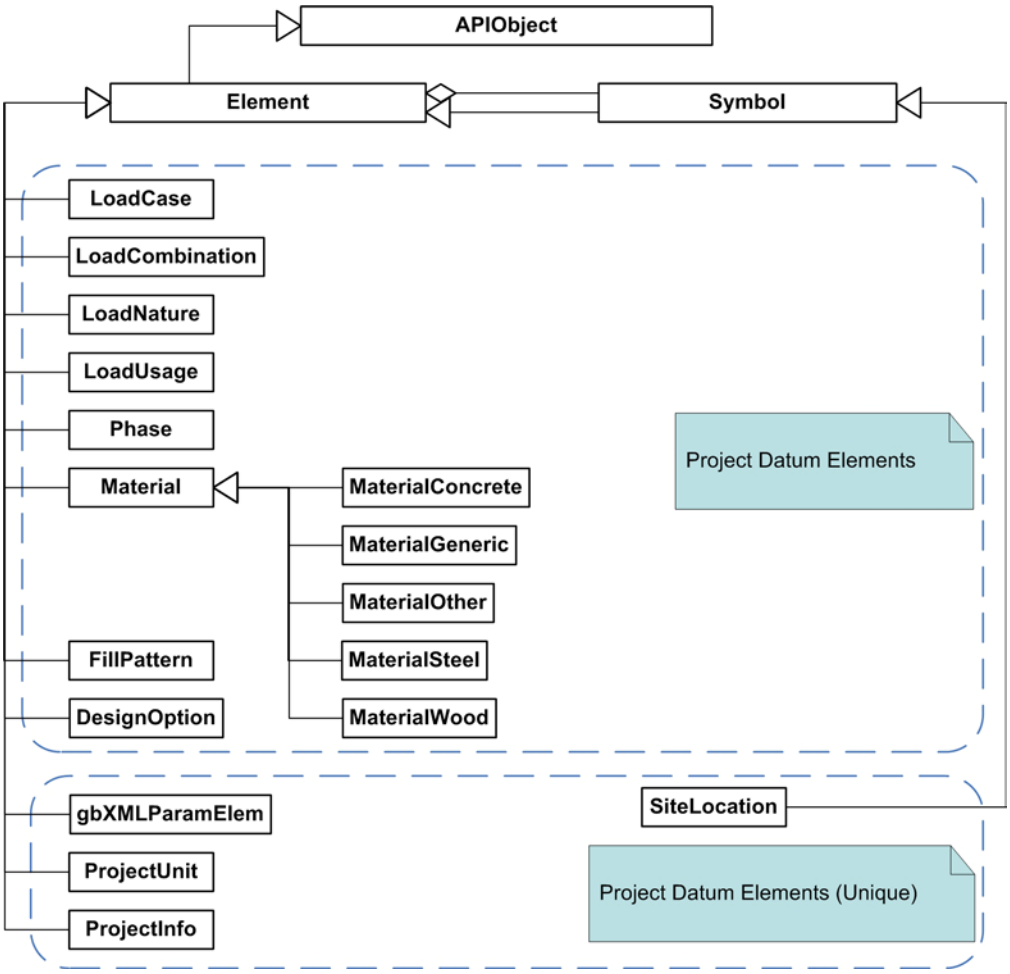



**Figure 36: Information Elements diagram**

## 5.2 Other Classifications

Elements are also classified by the following:

- Category
- Family
- Symbol
- Instance

There are some relationships between the classifications. For example:

- Distinguish different kinds of FamilyInstances by the category. Items such as structural columns are in the Structural Columns category, beams and braces are in the Structural Framing category, and so on.
- Differentiate structural FamilyInstance Elements by the symbol.

### 5.2.1 Categories

Category represents the category or subcategory to which an Element belongs. It is used to identify the inferred Element type. For example, anything in the walls Category is considered a wall. Other categories include doors and rooms.

Categories are the most general class. The Categories object is a map that contains all Category objects in the document and is subdivided into the following:

- Model Categories - Model Categories include beams, columns, doors, windows, and walls.
- Annotation Categories. Annotation Categories include dimensions, grids, levels, and textnotes.

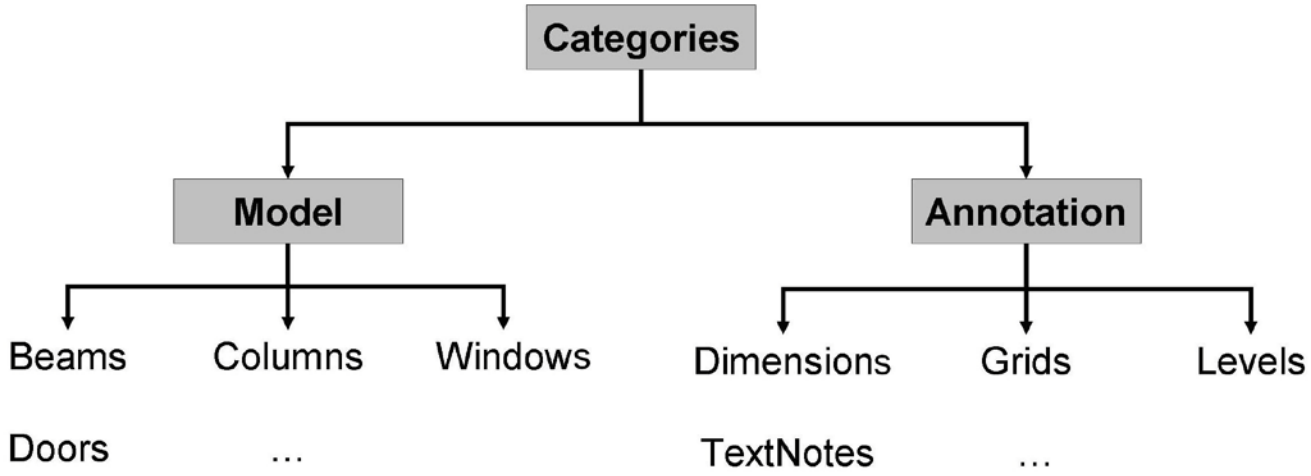



**Figure 37: Categories**

**Note:** The following guidelines apply to categories:

- In general, the following rules apply to categories:
  - Each family object belongs to a category
  - Non-family objects, like materials and views, do not belong to a category
  - There are exceptions such as ProjectInfo, which belongs to the Project Information category.
- An element and its corresponding symbols are usually in the same category. For example, a basic wall and its wall type Generic – 8" are all in the Walls category.
- The same type of Elements can be in different categories. For example, SpotDimensions has the SpotDimensionType, but it can belong to two different categories: Spot Elevations and Spot Coordinates.
- Different Elements can be in the same category because of their similarity or for architectural reasons. ModelLine and DetailLine are in the Lines category.

To gain access to all Revit categories in a document's Setting class, use one of the following techniques:

- Get the Categories from the document properties.
- Get a specific category quickly from the categories map using the BuiltInCategory enumerated type.

**Code Region 1-Elements Essentials**

```
Document doc = commandData.Application.ActiveDocument;
// Get settings of current document
Settings documentSettings = doc.Settings;
// Get all categories of current document
Categories groups = documentSettings.Categories;
// get Floor category according to OST_Floors and show its name
Category floorCategory = groups.get_Item(BuiltInCategory.OST_Floors);
```

Category is used in the following manner:

- Category is used to classify elements. The element category determines certain behaviors. For example, all elements in the same category can be included in the same schedule.
- Elements have parameters based on their categories.
- Categories are also used for controlling visibility and graphical appearance in Revit.

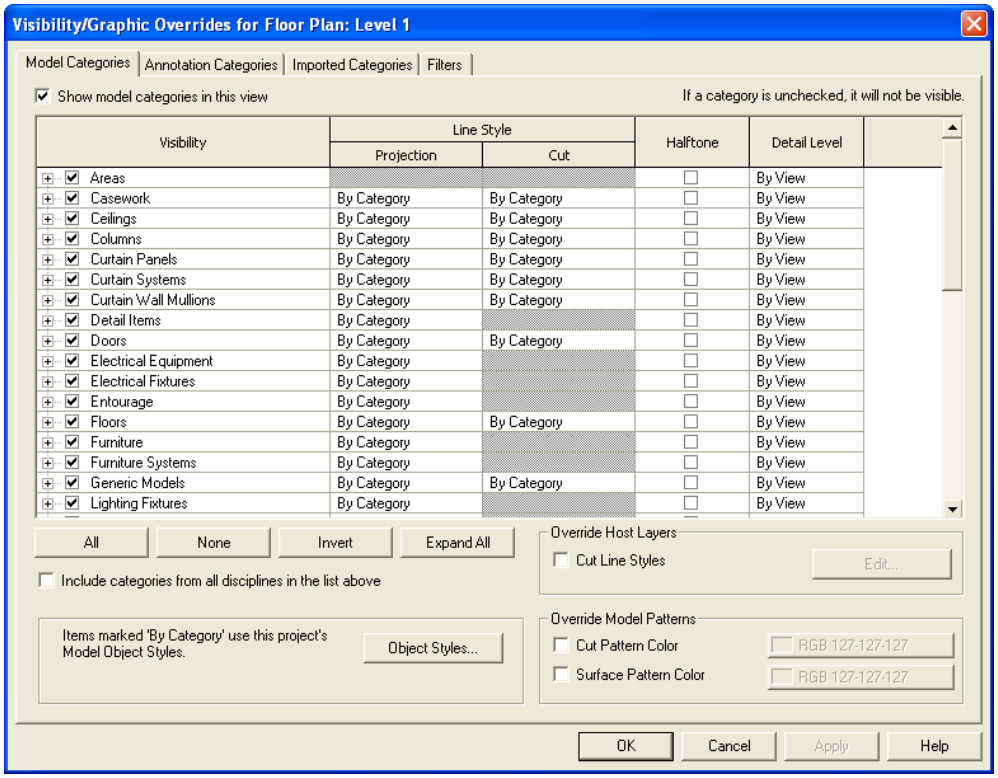



**Figure 37: Visibility by Category**

An element's category is determined by the Category ID.

- Category IDs are represented by the ElementId class.
- Imported Category IDs correspond to elements in the document.
- Most categories are built-in and their IDs are constants stored in ElementIds.
- Each built-in category ID has a corresponding value in BuiltInCategory Enumeration. They can be converted to corresponding BuiltInCategory enumerated types.
- If the category is not built-in, the ID is converted to a null value.

**Code Region 2-Elements Essentials**

```
Document doc = commandData.Application.ActiveDocument;
ElementIterator iter = doc.Elements;
iter.Reset();
BuiltInCategory enumCategory = BuiltInCategory.INVALID;
while (iter.MoveNext())
{
     Autodesk.Revit.Element element = iter.Current as Autodesk.Revit.Element;
     if (null != element && null != element.Category)
     {
```

```
            enumCategory = (BuiltInCategory)element.Category.Id.Value;
      }
}
```

**Note:** To avoid Globalization problems when using Category.Name, BuiltInCategory is a better choice. Category.Name can be different in different languages

### 5.2.2 Family

Families are classes of Elements within a category. Families can group Elements by the following:

- A common set of parameters (properties).
- Identical use.
- Similar graphical representation.

Most families are component Family files, meaning that you can load them into your project or create them from Family templates. You determine the property set and the Family graphical representation.

Another family type is the system Family. System Families are not available for loading or creating. Revit predefines the system Family properties and graphical representation; they include walls, dimensions, roofs, floors (or slabs), and levels.

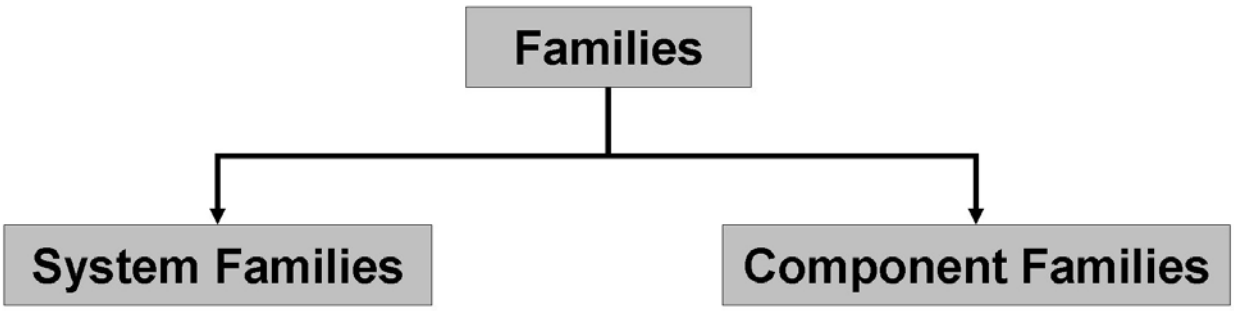



**Figure 38: Families**

In addition to functioning as an Element class, Family is also a template used to generate new items that belong to the Family.

#### 5.2.2.1 Family in the Revit API

In the Revit API, both the Family class and FamilyInstance belong to the Component Family. Other Elements include System Family.

Families in the Revit API are represented by three objects:

- Family
- FamilySymbol
- FamilyInstance.

Each object plays a significant role in the Family structure.

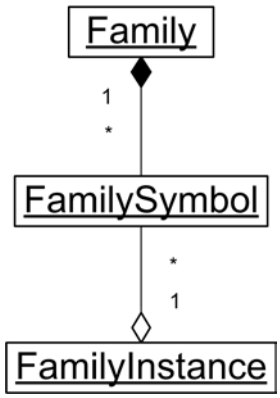



**Figure 39: Family, FamilySymbol and FamilyInstance UML diagram**

The Family object has the following characteristics:

- Represents an entire family such as a beam.
- Represents the entire family file on a disk.
- Contains a number of FamilySymbols.

The FamilySymbol object represents a specific set of family settings in the Family such as the Type, Concrete-Rectangular Beam: 16x32.

The FamilyInstance object is a FamilySymbol instance representing a single instance in the Revit project. For example, the FamilyInstance can be a single instance of a 16x32 Concrete-Rectangular Beam in the project.

**Note:** Remember that the FamilyInstance exists in Component Elements, Datum Elements, and Annotation Elements.

Consequently, the following rules apply:

- Each FamilyInstance has one FamilySymbol.
- Each FamilySymbol belongs to one Family.
- Each Family contains one or more FamilySymbols.

For more detailed information, see the Component Elements chapter.

### 5.2.3 Symbol

In the Revit API, Symbols are usually non-visible elements used to define instances. Symbols are called Types in the user interface.

- A type can be a specific size in a family, such as a 1730 X 2032 door, or an 8x4x1/2 angle.
- A type can be a style, such as default linear or default angular style for dimensions.

Symbols represent Elements that contain shared data for a set of similar elements. In some cases, Symbols represent building components that you can get from a warehouse, such as doors or windows, and can be placed many times in the same building. In other cases, Symbols contain host object parameters or other elements. For example, a WallType Symbol contains the thickness, number of layers, material for each layer, and other properties for a particular wall type.

FamilySymbol is a symbol in the API. It is also called Family Type in the Revit user interface. FamilySymbol is a class of elements in a family with the exact same values for all properties. For example, all 32x78 six-panel doors belong to one type, while all 24x80 six-panel doors belong to another type. Like a Family, a FamilySymbol is also a template. The FamilySymbol object is derived from the Symbol object and the Element object.

### 5.2.4 Instance

Instances are items with specific locations in the building (model instances) or on a drawing sheet (annotation instances). Instance represents transformed identical copies of a Symbol. For example, if a building contains 20 windows of a particular type, there is one Symbol with 20 Instances. Instances are called Components in the user interface.

**Note:** For FamilyInstance, it is better to use the Symbol property instead of the ObjectType property to get the corresponding FamilySymbol. It is convenient and safe since you do not need to do a type conversion.

## 5.3 Element Retrieval

Elements in Revit are very common. Retrieving the elements that you want from Revit is necessary before using the API for any Element command.

### 5.3.1 General

In general, an element is retrieved from the document Elements collection object using ElementId or iteration. The following figure shows how to get the element using the Revit application.

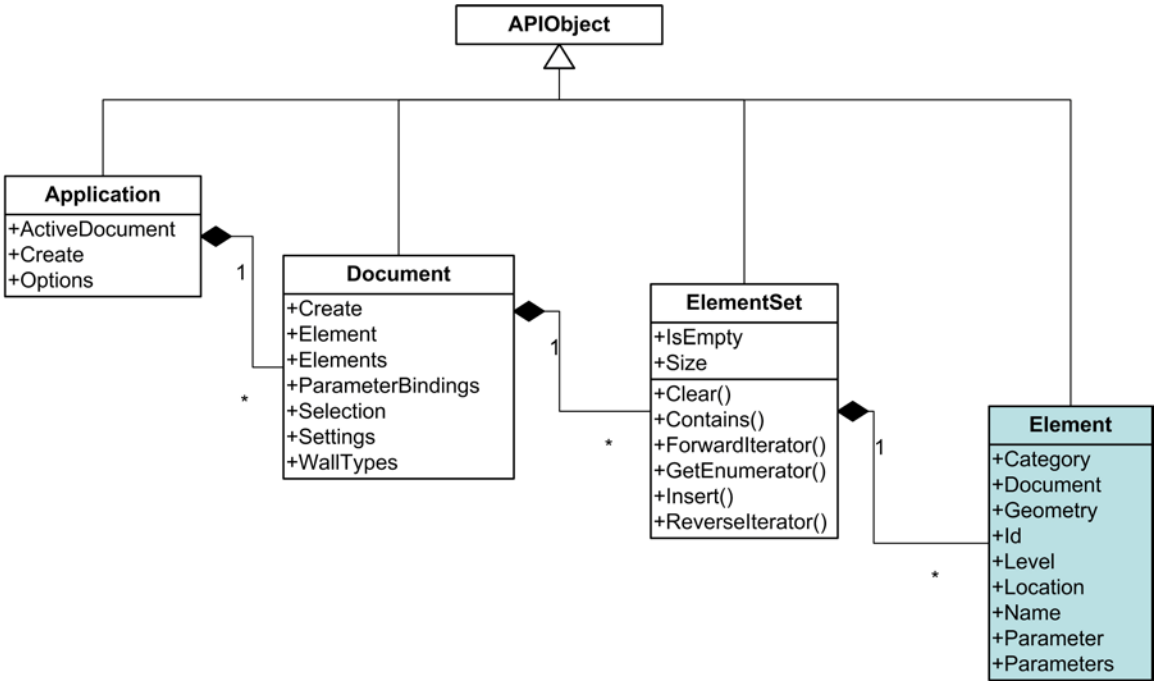



**Figure 40: Retrieving Elements diagram**

- Application refers to the Autodesk.Revit.Application, providing access to documents, options, and other application-wide data and settings. Gain access to the application from the parameter commandData in the IExternalCommand.Execute method.
- Document represents an open Revit project.
- ElementSet is a set that contains element objects.
- All Elements are common elements, including many abstract and invisible objects.

You can get all current project objects from the active document using the Application.ActiveDocument.Elements property.

### 5.3.2 Selection

Rather than getting all of the elements in the model, you can access just the elements that have been selected by the users. You can get the selected objects from the current active document using the Application.ActiveDocument.Selection.Elements property. The selected objects are in an ElementSet in Revit. From this Element set, all selected Elements are retrieved.

The selected Element set can be changed. To modify the Selection.Elements:

1. Create a new SelElementSet.
2. Put Elements in it.
3. Set the Selection.Elements to the new SelElementSet instance.

The following example illustrates how to change the selected Elements.

**Code Region 3-Elements Essentials**

```
Document doc = commandData.Application.ActiveDocument;
Selection choices = doc.Selection;
SelElementSet collection = choices.Elements;

//Create a Column
FamilyInstance newColumn = CreateColumn(doc);
//Create a new SelElementSet
SelElementSet newSelectedElementSet = commandData.Application.Create.NewSelElementSet();
//Create a new SelElementSet as below is ok
//SelElementSet newSelectedElementSet = new SelElementSet();

//Reassemble the selected elements
newSelectedElementSet.Add(newColumn);
//Reset the selected elements.
choices.Elements = newSelectedElementSet;
```

If you want to add one or more user-specified Elements to the set, use the PickOne and WindowSelect methods in the Selection class. This allows the user to select one or more Elements using the cursor and then returns control to your application.

- The PickOne method adds one more Element to the set if the Element selected differs from the Elements already in the set.
- The WindowSelect method adds the selected Elements to the set if possible.

**Code Region 4-Elements Essentials**

```
Document doc = commandData.Application.ActiveDocument;
Selection choices = doc.Selection;
// Pick one object from Revit.
bool hasPickOne = choices.PickOne();
// Choose objects from Revit.
bool hasPickSome = choices.WindowSelect();
```

The selected Elements and all Elements do not use the same collection class. You must write two separate functions depending on whether all Elements or selected Elements are needed.

For more information about retrieving Elements from selected Elements and all Elements, see the Walkthrough: Retrieve Selected Elements and Walkthrough: Retrieve All Elements sections in the Getting Started chapter.

### 5.3.3 Other

In addition to using the general way to access Elements, the Revit API has properties in the Document class to get the specified Elements from the current active document without iterating all Elements. The specified Elements you can retrieve are listed in the following table.

| Element | Access in property of Document |
|---|---|
| ProjectInfo | Document.ProjectInformation |
| ProjectUnit | Document.ProjectUnit |
| ProjectLocation | Document.ProjectLocations<br>Document.ActiveProjectLocation |
| SiteLocation | Document.SiteLocation |
| Phase | Document.Phases |
| FillPattern | Document.Settings.FillPatterns |
| Material | Document.Settings.Materials |
| FamilySymbol relative to the deck profile of the layer within a structure | Document.DeckProfiles |
| FamilySymbol in the Title Block category | Document.TitleBlocks |
| All Element types | Document.AnnotationSymbolTypes / BeamSystemTypes / ContFootingTypes / DimensionTypes / FloorTypes / GridTypes / LevelTypes / RebarBarTypes / RebarHookTypes / RoomTagTypes / SpotDimensionTypes / WallTypes / TextNoteTypes |

**Table 10: Retrieving Elements from document properties**

## 5.4 General Properties

The following properties are common to each Element created using Revit.

### 5.4.1 ElementId

Every element in an active document has a unique identifier represented by the ElementId storage type. ElementId objects are project wide. It is a unique number that is never changed in the element model, which allows it to be stored externally to retrieve the element when needed.

To view an element ID in Revit, complete the following steps:

1. From the Tools menu, select Element Ids. The Element Ids submenu appears.
2. Select Ids of Selection to get the ID number for one element.

**Figure 41: ElementId**

In the Revit API, you can create an ElementId directly, and then associate a unique integer value to the new ElementId. The new ElementId value is 0 by default.

**Code Region 5-Elements Essentials**

```
ElementId id = new ElementId();
id.Value = 155507;
```

ElementId has the following uses:

- Use ElementId to retrieve a specific element from Revit. From the Revit Application class, gain access to the active document, and then get the specified element using the Document.Element(ElementId) property.

**Code Region 6-Elements Essentials**

```
Document doc = commandData.Application.ActiveDocument;
ElementId id = new ElementId();
id.Value = 155507;
// Get specified elements using ElementId.
Element element = doc.get_Element(ref id);
```

If the ID number does not exist in the project, the element you retrieve is null.

- Use ElementId to check whether two Elements in one project are equal or not. It is not recommended to use the Object.Equal method.

### 5.4.2 UniqueId

Every element has a UniqueId, represented by the String storage type. The UniqueId corresponds to the ElementId. However, unlike ElementId, UniqueId functions like a GUID (Globally Unique Identifier), which is unique across separate Revit projects. UniqueId can help you to track elements when you export Revit project files to other formats.

**Code Region 7-Elements Essentials**

```
String uniqueId = element.UniqueId;
```

**Note:** The ElementId is only unique in the current project. It is not unique across separate Revit projects. UniqueId is always unique across separate projects.

### 5.4.3 Location

The location of an object is important in the building modeling process. In Revit, some objects have a point location. For example a table has a point location. Other objects have a line location, representing a location curve or no location at all. A wall is an element that has a line location.

The Revit API provides the Location class and location functionality for most elements. For example, it has the Move and Rotate methods to translate and rotate the elements. However, the Location class has no property from which you can get information such as a coordinate. In this situation, downcast the Location object to its subclass—like LocationPoint or LocationCurve—for more detailed location information and control using object derivatives. The following picture shows the inherited relationship.

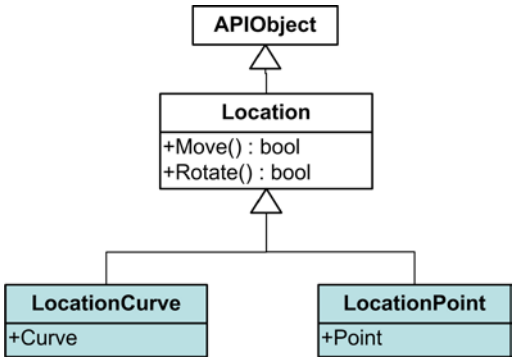



**Figure 42: Location diagram**

Retrieving an element's physical location in a project is useful when you get the geometry of an object. The following rules apply when you retrieve a location:

- Wall, Beam, and Brace are curve-driven using LocationCurve.
- Room, RoomTag, SpotDimension, Group, FamilyInstances that are not curve-driven, and all In-Place-FamilyInstances use LocationPoint.

In the Revit API, curve-driven means that the geometry or location of an element is determined by one or more associated curves. Almost all analytical model elements are curve-driven – linear and area loads, walls, framing elements, and so on.

Other Elements cannot retrieve a LocationCurve or LocationPoint. They return Location with no information.

| Location Information | Elements |
|---|---|
| LocationCurve | Wall, Beam, Brace, Structural Truss, LineLoad(without host) |
| LocationPoint | Room, RoomTag, SpotDimensioin, Group, Column, Mass |
| Only Location | Level, Floor, some Tags, BeamSystem, Rebar, Reinforcement, PointLoad, AreaLoad(without Host), Span Direction(IndependentTag) |
| No Location | View, LineLoad(with host), AreaLoad(with Host), BoundaryCondition |

**Table 11: Elements Location Information**

**Note:** There are other Elements without Location information. For example a LineLoad (with host) or an AreaLoad (with host) have no Location.

Some FamilyInstance LocationPoints, such as all in-place-FamilyInstances and masses, are specified to point (0, 0, 0) when they are created. The LocationPoint coordinate is changed if you transform or move the instance.

To change a Group's LocationPoint, do one of the following:

- Drag the Group origin in the Revit UI to change the LocationPoint coordinate. In this situation, the Group LocationPoint is changed while the Group's location is not changed.
- Move the Group using the Document.Move method to change the LocationPoint. This changes both the Group location and the LocationPoint.

For more information about LocationCurve and LocationPoint, see the Editing Elements chapter Move section.

### 5.4.4 Level

Levels are finite horizontal planes that act as a reference for level-hosted or level-based elements, such as roofs, floors, and ceilings. The Revit API provides a Level class to represent level lines in Revit. Get the Level object to which the element is assigned using the API if the element is level-based.

**Code Region 8-Elements Essentials**

```
// Get the level object to which the element is assigned.
Level level = element.Level;
```

A number of elements, such as a column, use a level as a basic reference. When you get the column level, the level you retrieve is the Base Level.

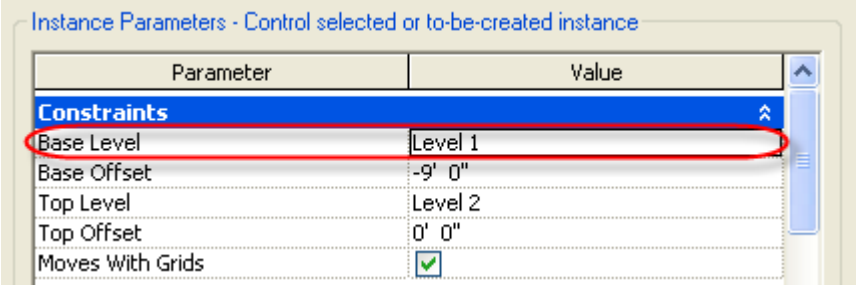



**Figure 43: Column Base Level parameter**

**Note:** Get the Beam or Brace level using the Reference Level parameter. From the Level property, you only get null instead of the reference level information.

Level is the most commonly used element in Revit. In the Revit API, all levels in the project are located by iterating over the entire project and searching for Elements.Level objects. The following code retrieves all levels from the current active document.

**Code Region 9-Elements Essentials**

```
Document doc = commandData.Application.ActiveDocument;
List<Level> levelList = new List<Level>();
ElementFilterIterator iter = doc.get_Elements(typeof(Level));
iter.Reset();
//Retrieve all levels within the project
while (iter.MoveNext())
{
   levelList.Add(iter.Current as Level);
}
```

For more Level details, see the Datum and Information Elements chapter.

### 5.4.5 Parameter

Every element has a set of parameters that users can view and edit in Revit. The parameters are visible in the Element Properties dialog box (select any element and click the Properties button next to the type selector). For example, the following image shows Room parameters.

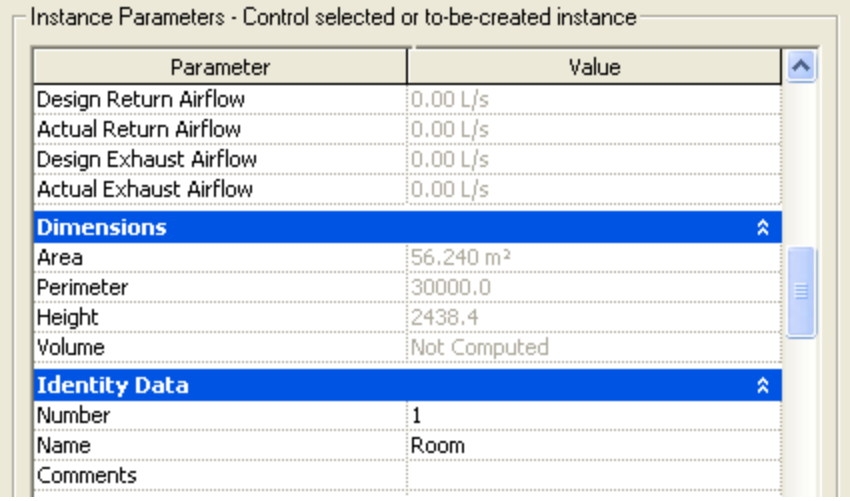



**Figure 44: Room parameters**

In the Revit API, each Element object has a Parameters property, which is a collection of all the properties attached to the Element. You can change the property values in the collection. For example, you can get the area of a room from the room object parameters; additionally, you can set the room number using the room object parameters. The Parameter is another way to provide access to property information not exposed in the element object.

In general, every element parameter has an associated parameter ID. Parameter IDs are represented by the ElementId class. For user-created parameters, the IDs correspond to real elements in the document. However, most parameters are built-in and their IDs are constants stored in ElementIds.

Parameter is a generic form of data storage in elements. In the Revit API, it is best to use the built-in parameter ID to get the parameter. Revit has a large number of built-in parameters available using the BuiltInParameter enumerated type.

For more details, see the Parameter Chapter.

# 6 Parameter

Revit provides a general mechanism for giving each element a set of parameters that you can edit. Parameters are visible in the Element Properties dialog box. This chapter describes how to get and use parameters using the Revit API.

In the Revit API, Parameters are managed in the Element class by a complex mechanism and set of rules. Learn how to use the Revit user interface for adding and managing parameters before using the Revit API. The parameter user interface components are located in the following locations:

- Element Properties dialog box
- Shared Parameters dialog box (from the File menu),
- Project Parameters dialog box (from the Settings menu),
- Family Types dialog box (from the Settings menu when editing a family).

Retrieve the Parameter object from Element if you know the built-in ID, definition, or shared parameter GUID (Global Unique Identity Code).

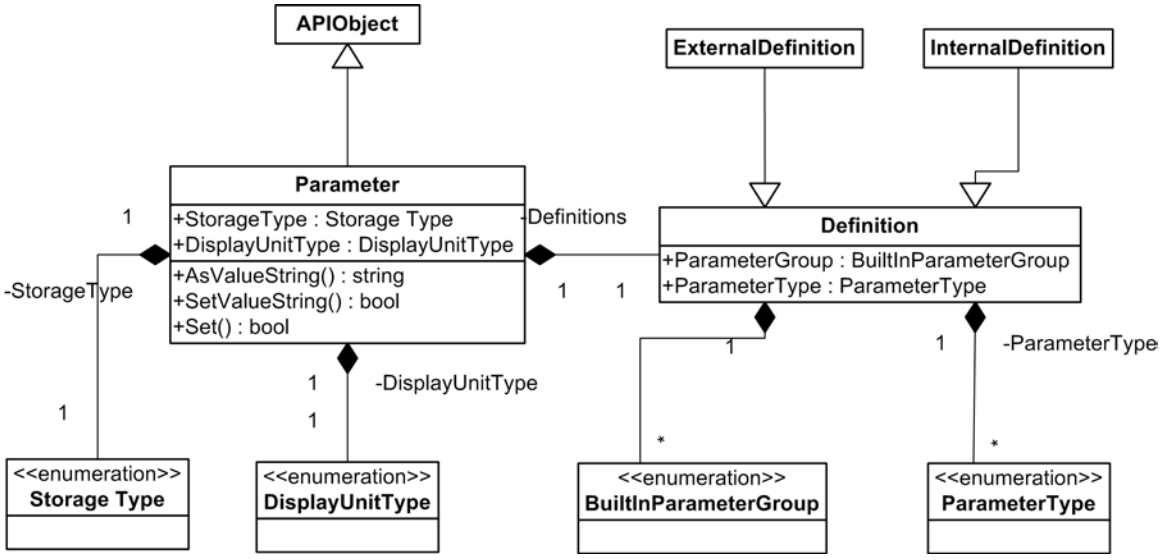



**Figure 45: Parameter-related classes diagram**

## 6.1 Walkthrough: Get Selected Element Parameters

The Element Parameters are retrieved by iterating through the Element ParameterSet. The following code sample illustrates how to retrieve the Parameter from a selected element:

**Code region 1 - Parameter**

```
public IExternalCommand.Result Execute(ExternalCommandData commandData, ref string message,
  ElementSet elements)
{
   Document doc = commandData.Application.ActiveDocument;
    StringBuilder st = new StringBuilder();

   SelElementSet selElements = doc.Selection.Elements;
    Autodesk.Revit.Element selectedElem = null;
    foreach (Autodesk.Revit.Element elem in selElements)
   {
```

```
            selectedElem = elem;
            break;
        }

        // iterate element's parameter set
        foreach (Parameter para in selectedElem.Parameters)
        {
            st.AppendLine(GetParameterInformation(para, doc));
        }

        // Give the user some information
        MessageBox.Show(st.ToString(), "Revit", MessageBoxButtons.OK);

        // If everything goes well, return succeed.
        return IExternalCommand.Result.Succeeded;
    }
    String GetParameterInformation(Parameter para, Document document)
    {
        string defName = para.Definition.Name;
        string val;
        // Use different method to get parameter according to the storage type
        switch (para.StorageType)
        {
            case StorageType.Double:
                //covert the number into Metric
                return defName + " : " + para.AsValueString();
            case StorageType.ElementId:
                //find out the name of the element
                ElementId id = para.AsElementId();
                if (id.Value >= 0)
                {
                    return defName + " : " + document.get_Element(ref id).Name;
                }
                else
                {
                    return defName + " : " + id.Value.ToString();
                }
            case StorageType.Integer:
                if (ParameterType.YesNo == para.Definition.ParameterType)
                {
                    if (para.AsInteger() == 0)
                    {
                        return defName + " : " + "False";
                    }
                    else
```

```
            {
               return defName + " : " + "True";
            }
         }
         else
         {
            val = para.AsInteger().ToString();
            return defName + " : " + val;
         }
      case StorageType.String:
         return defName + " : " + para.AsString();
      default:
         return "Unexposed parameter.";
   }
}
```

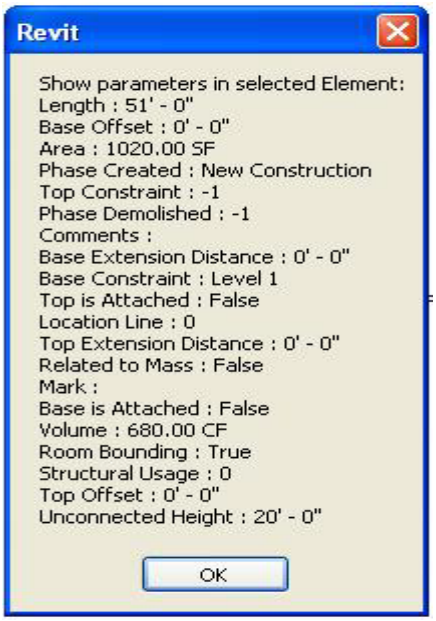

Revit
Show parameters in selected Element:
Length : 51' - 0"
Base Offset : 0' - 0"
Area : 1020.00 SF
Phase Created : New Construction
Top Constraint : -1
Phase Demolished : -1
Comments :
Base Extension Distance : 0' - 0"
Base Constraint : Level 1
Top is Attached : False
Location Line : 0
Top Extension Distance : 0' - 0"
Related to Mass : False
Mark :
Base is Attached : False
Volume : 680.00 CF
Room Bounding : True
Structural Usage : 0
Top Offset : 0' - 0"
Unconnected Height : 20' - 0"
OK

**Figure 46: Get wall parameters result**

**Note:** In Revit, some parameters have values in the drop-down list in the Element Properties dialog box. You can get the numeric values using the Revit API, but you cannot get the string representation for the values using the Parameter.AsValueString method.

## 6.2 Definition

The Definition object describes the data type, name, and other Parameter details. There are two kinds of definition objects derived from this object.

- InternalDefinition represents all kinds of definitions existing entirely in the Revit database.
- ExternalDefinition represents definitions stored on disk in a shared parameter file.

You should write the code to use the Definition base class so that the code is applicable to both internal and external parameter Definitions. The following code sample shows how to find a specific parameter using the definition name.

**Code region 2 - Parameter**

```
//Find parameter using the Parameter's definition name.
public Parameter FindParameter(ParameterSet parameters)
{
   Parameter foundParameter = null;
   //In the Revit English edition, you can find the parameter
   //name, Base Offset. However, in other language editions,
   //you may need to change this string value.
   string name = "Base Offset";
   foreach (Parameter parameter in parameters)
   {
      if (parameter.Definition.Name == name)
      {
         foundParameter = parameter;
         break;
      }
   }
   return foundParameter;
}
```

**Note:** This example shows how to get Element parameters using the parameter definition names. Base Offset is the definition name of a parameter in Revit (English Edition). You can change the string values based on the language edition you use.

### 6.2.1 ParameterType

This property returns the visible interpretation of the parameter data. The parameter data is stored in the following:

- Integer
- Double
- String
- An ElementId

For Revit users, the value appears as Length, Volume, Material, and so on.

For more details about ParameterType.Material, please refer to the Material chapter.

### 6.2.2 ParameterGroup

The Definition class ParameterGroup property returns the parameter definition group ID. The BuiltInParameterGroup is an enumerated type listing all built-in parameter groups supported by Revit. Parameter groups are used to sort parameters in the Element Properties dialog box.

## 6.3 BuiltInParameter

The Revit API has a large number of built-in parameters. The parameter ID is used to retrieve the specific parameter from an element, if it exists, using the Element.Parameter property. However, not all parameters can be retrieved using the ID. For example, family parameters are not exposed in the Revit API, therefore, you cannot get them using the built-in parameter ID.

The following code sample shows how to get the specific parameter using the BuiltInParameter Id:

**Code region 3 - Parameter**

```
public Parameter FindWithBuildingParameterID(Wall wall)
{
    //Use the WALL_BASE_OFFSET paramametId
  // to get the base offset parameter of the wall.
   BuiltInParameter paraIndex = BuiltInParameter.WALL_BASE_OFFSET;
   Parameter parameter = wall.get_Parameter(paraIndex);

   return parameter;
}
```

**Note:** With the get_Parameter method, use an Enumerated type BuiltInParameter as the method parameter. For example, use BuiltInParameter.GENERIC_WIDTH.

If you do not know the exact BuiltInParameter ID, get the parameter by iterating the ParameterSet collection.

Another approach for testing or identification purposes is to test each BuiltInParameter using the get_Parameter method. Use the Enum class to iterate as illustrated in the following code. When you use this method, it is possible that the ParameterSet collection may not contain all parameters returned from the get_Parameter method, though this is infrequent.

**Code region 4 - Parameter**

```
public void FindBuiltInParameter(Wall wall)
{
   foreach (BuiltInParameter bip in Enum.GetValues(typeof(BuiltInParameter)))
   {
      Parameter param = wall.get_Parameter(bip);
      if (null != param)
      {
         // Do something to see what you have
      }
   }
}
```

## 6.4 StorageType

StorageType describes the type of parameter values stored internally. The property returns one of the following types:

- String
- Integer
- Double
- ElementId

Based on the property value, use the corresponding access and Set methods to retrieve and set the parameter data value.

The StorageType is an enumerated type that lists all internal parameter data storage types supported by Revit:

| Member Name | Description |
|---|---|
| **String** | Internal data is stored as a string of characters. |
| **ElementId** | Data type represents an element and is stored as an Element ID. |
| **Double** | Data is stored internally as an 8-byte floating point number. |
| **Integer** | Internal data is stored as a signed 32-bit integer. |
| **None** | None represents an invalid storage type. Do not use this value. |

**Table 12: Storage Type**

In most cases, the ElementId value is a positive number. However, it can be a negative number. When the ElementId value is negative, it does not represent an Element but some other thing. For example, the storage type parameter for a beam's Vertical Projection is ElementId. When the parameter value is Level 1 or Level 2, the ElementId value is positive and corresponds to the ElementId of that level. However, when the parameter value is set to Auto-detect, Center of Beam or Top of Beam, the ElementId value is negative.

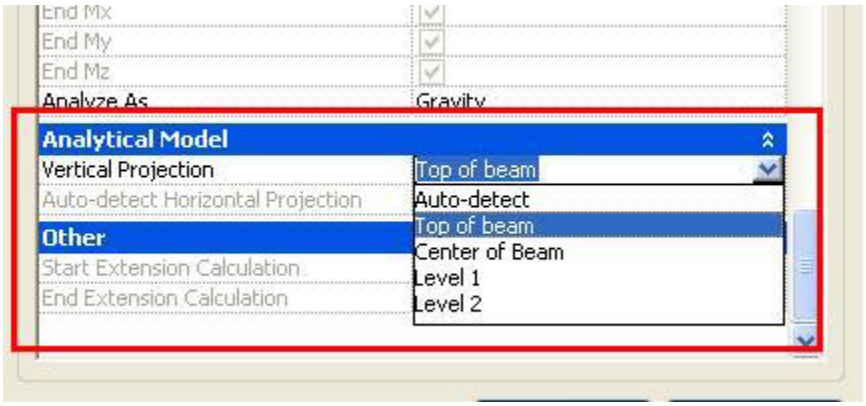



**Figure 47: Storage type sample**

The following code sample shows how to change a parameter value:

**Code region 5 - Parameter**

```
public bool SetParameter(double value)
{
   //…
   //Try to get the select wall in Revit.
   Wall wall = GetWall();
   if (null != wall)
   {
      // Use the paramametId WALL_BASE_OFFSET to get the
     //"Base Offset" parameter of the wall.
      BuiltInParameter paraIndex = BuiltInParameter.WALL_BASE_OFFSET;
      Parameter foundParameter = wall.get_Parameter(paraIndex);

      //if the parameter is readonly, you can't change the value of it
      if (null != foundParameter && !foundParameter.IsReadOnly )
```

```
      {
         StorageType parameterType = foundParameter.StorageType;
         if (StorageType.Double != parameterType)
         {
            throw new Exception("The types of value and parameter are different!");
         }
         //If successful, the result is true
         bool result = foundParameter.Set(value);
         return result;
      }
   }
   return false;
}
```

The Set method return value indicates that the Parameter value was changed. It does not indicate whether the change was successful or not. The Set method returns true if the Parameter value was changed, otherwise it returns false.

Not all Parameters are writable. An Exception is thrown if the Parameter is read-only.

## 6.5 AsValueString and SetValueString

The AsValueString method and SetValueString method are Parameter class methods. The two methods are only applied to value type parameters.

Use the AsValueString method to get the parameter value as a string with the unit of measure. For example, the Base Offset value, a wall parameter, is a Double value. Usually the value is shown as a string like -20’0” in the Element Properties dialog box:

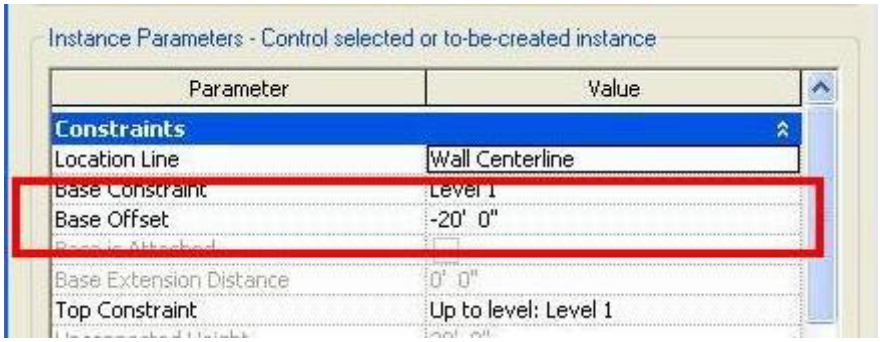



**Figure 48: AsValueString and SetValueString sample**

Using the AsValueString method, you get the -20’0” string value directly. Otherwise you get a double value like -20 without the unit of measure if you use the AsDouble method.

Use the SetValueString method to change the value of a value type parameter instead of using the Set method. The following code sample illustrates how to change the parameter value using the SetValueString method:

**Code region 6 - Parameter**

```
public bool SetWithValueString(Parameter foundParameter)
{
   bool result = false;
   if (!foundParameter.IsReadOnly)
```

```
   {
      //If successful, the result is true
      result = foundParameter.SetValueString("-22\'3\"");
   }
   return result;
}
```

## 6.6 Parameter Tips and Tricks

There are several relationships between Parameters. In the Property dialog box, there are some combo box parameters that appear to be a drop-down list with each choice shown as a string. Internally, each choice has an associated value, either an Integer or an ElementId. All values for a given combo box parameter share the same data type.

1. In the Element Property dialog box, a combo box is selected, disabling other parameters. For example:
   - Select a beam in Revit then open the Element Property dialog box. If the Start Release is set to Pinned, the parameters Start Fx, Start Fy, and other parameters are disabled.

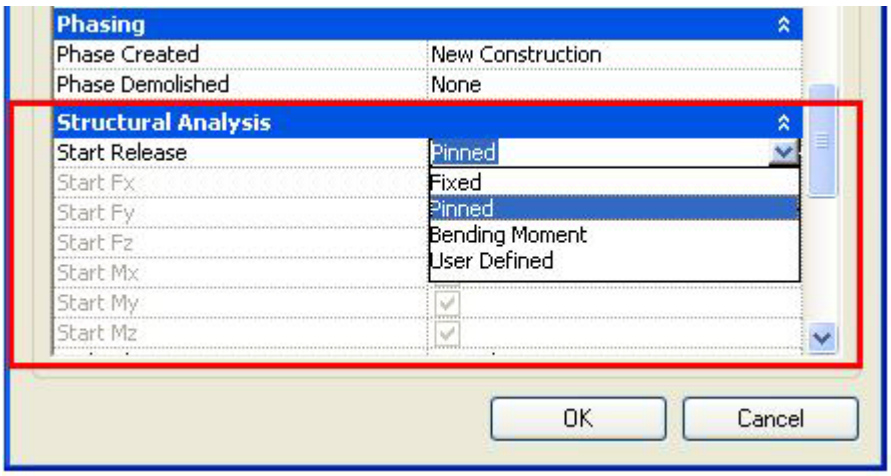



**Figure 49: Parameter relationship sample**

   - If the Start Release is set to User Defined, the parameters Start Fx, Start Fy, and other parameters are editable.

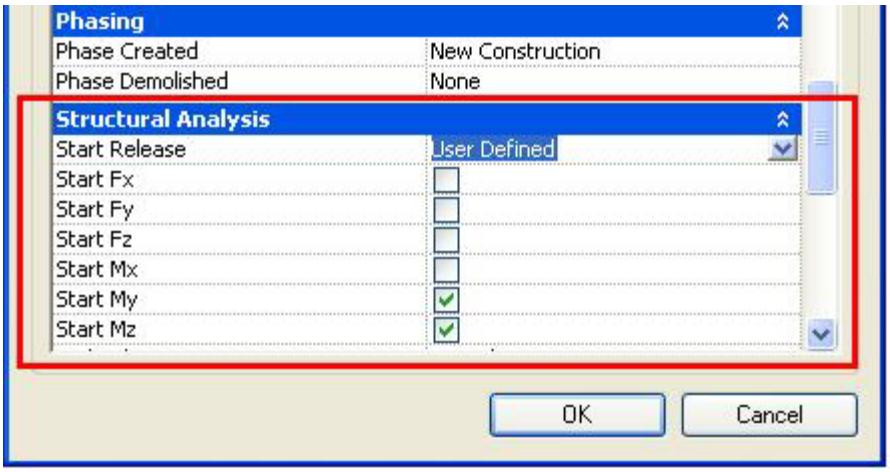



**Figure 50: Parameter relationship sample**

2. Some elements have different parameters when the value of a particular parameter is changed.
   - Select a beam system then set the Layout Rule to Fixed Distance. In this case, a parameter named Fixed Spacing appears in the property dialog box.

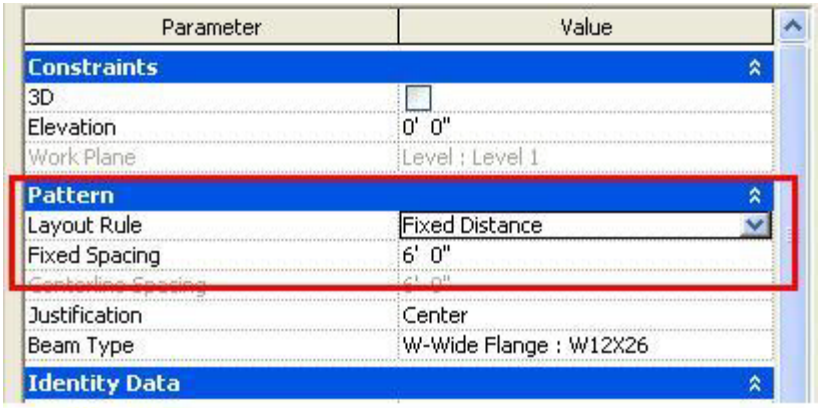



**Figure 51: Parameter relationship sample**

   - When the Layout Rule is set to Fixed Number, the Number of Lines parameter appears instead of Fixed Spacing.

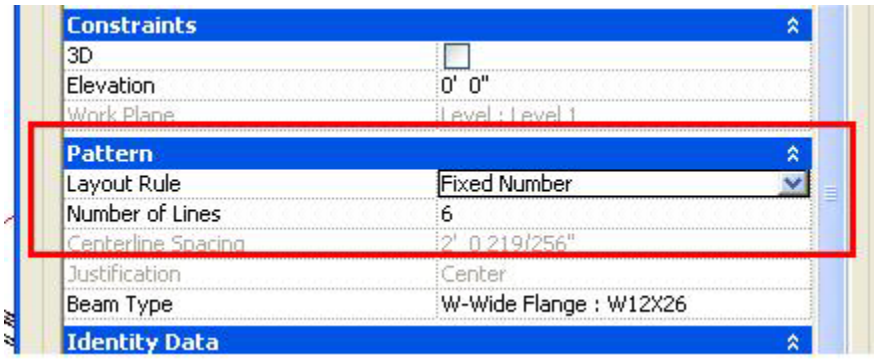



**Figure 52: Parameter relationship sample**

3. Some parameters are computed in Revit, such as wall Length and Area parameter. These parameters are always read-only because they depend on the element's internal state.

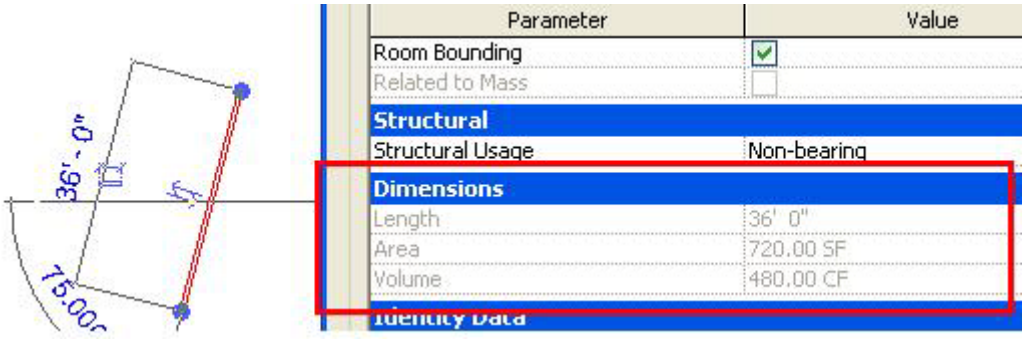



**Figure 53: Wall computed parameters**

It is vital to pay attention to the relationships between parameters when working on parameters using the Revit API.

## 6.7 Add Parameters to Elements

In the Revit API, you can use all of the defined parameters and you can add custom parameters that you define using the Revit user interface and the Revit API.

Users and developers using the Revit API can add custom parameters as shared parameters.

Complete the following steps to add a parameter to an element using the Revit API:

1. Open the shared parameters file using the Document.OpenSharedParameterFile method.
2. Use an existing group or create a new one, using the DefinitionFile.Groups property.
3. Use an existing external parameter definition or create a new one, using the DefinitionGroup.Definitions property.
4. Create a new Binding object using an InstanceBinding or a TypeBinding object.
5. Add the binding and definition to the document using the Document.ParameterBindings object.

For more details, refer to the Shared Parameter chapter.

# 7 Collection

The Revit API has some utility collection types such as Array, Map, Set collections, and related Iterators. There are many advantages to using Revit API Array, Map, and Set.

- They are generic, convenient, type safe, and fast to use.
- Although there are generic classes such as List<T> and Collection<T>in the .NET Framework 2.0 library, the API collection classes are closed to internal implementation making them faster.
- The method return value, the method input parameter, and properties are API collection classes, avoiding transformation between different collections. The collections inheritance hierarchy is as follows:

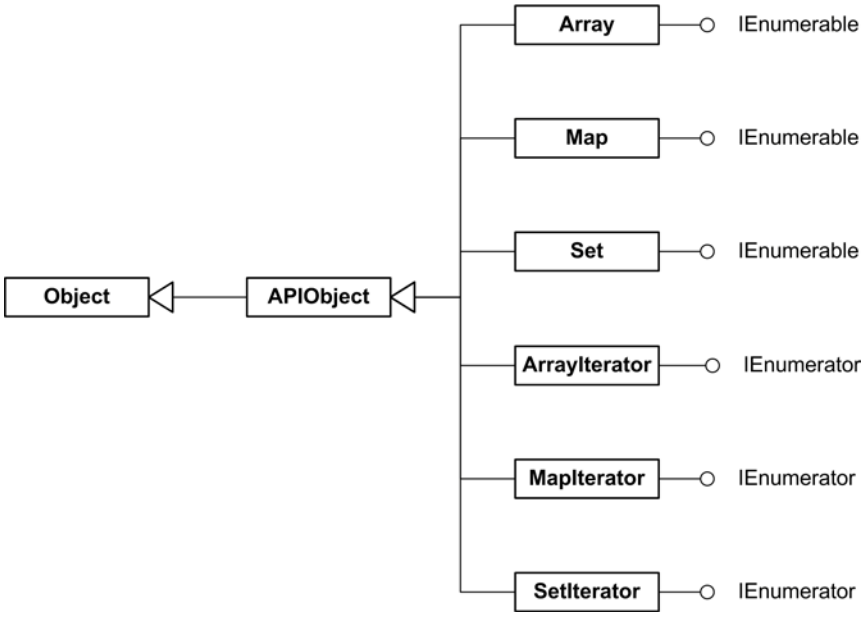



**Figure 54: Collection diagram**

**Note:** Array, Map, and Set in the Revit API are different from the classes in the .NET Framework Library such as System.Array. In the Revit API, they only implement interface IEnumerable but not IList, ICollection, ICloneable and so on. These collection types support the foreach semantics in C# (for each in C++/CLI, For Each in Visual Basic).

The IEnumerable and IEnumerator interfaces implemented in Revit collection types are defined in the System.Collection namespace.

## 7.1 Interface

The following sections discuss interface-related collection types.

### 7.1.1 IEnumerable

The IEnumerable interface is in the System.Collections namespace. It exposes the enumerator, which supports a simple iteration over a non-generic collection. The GetEnumerator method gets an enumerator that implements this interface. The returned IEnumerator object is iterated throughout the collection.

**Note:** The GetEnumerator method return value is an IEnumerator object. Consequently, the As operator is used to convert the object type.

### 7.1.2 IEnumerator

The IEnumerator interface is in the System.Collections namespace. It supports a simple iteration over a non-generic collection. IEnumerator is the base interface for all non-generic enumerators. The foreach statement in C# (For Each in Visual Basic) hides the enumerator's complexity.

**Note:** Using foreach is recommended instead of directly manipulating the enumerator.

Enumerators are used to read the collection data, but they cannot be used to modify the underlying collection.

Use IEnumerator as follows:

- Initially, the enumerator is positioned in front of the first element in the collection.
  - The Reset method moves the enumerator back to the original position. At this position, calling the Current property throws an exception.
  - Call the MoveNext method to advance the enumerator to the collection's first element before reading the current iterator value.
- The Current property returns the same object until either the MoveNext method or Reset method is called. The MoveNext method sets the current iterator to the next element.
- If MoveNext passes the end of the collection, the enumerator is positioned after the last element in the collection and MoveNext returns false.
  - When the enumerator is in this position, subsequent calls to the MoveNext also return false.
  - If the last call to the MoveNext returns false, calling the Current property throws an exception.
  - To set the current iterator to the first element in the collection again, call the Reset method followed by MoveNext.
- An enumerator remains valid as long as the collection remains unchanged.
  - If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is invalidated and the next call to the MoveNext or the Reset method throws an InvalidOperationException.
  - If the collection is modified between the MoveNext and the current iterator, the Current property returns to the specified element, even if the enumerator is already invalidated.

**Note:** All calls to the Reset method must result in the same state for the enumerator. The preferred implementation is to move the enumerator to the collection beginning, before the first element. This invalidates the enumerator if the collection is modified after the enumerator was created, which is consistent with the MoveNext and the Current properties.

### 7.1.3 As and Is Operators

The As operator is used to implement the conversion between two compatible reference types. For example:

```
string s = someObject as string;
if (null != s)
{
   // someObject is a string.
}
```

**Note:** The As operator is similar to a cast except that it yields null on conversion failure instead of raising an exception. For example, you can use the As operator in the following ways:

- someObject as string;

is equal to :

- someObject is string ? (string)someObject : (string)null;

Note that the expression in the first bullet is evaluated only once.

The As operator only performs reference conversions and boxing conversions. It cannot perform other conversions, such as user-defined conversions.

The Is operator is used to check whether the run-time object type is compatible with a particular type. For example, you can use the Is operator as follows:

```
if (obj is string) {  }
```

- An Is expression is true if the provided expression is non-null, and the provided object can be cast to the provided type without throwing an exception.
- The Is keyword results in a compile-time warning if the expression is always true or always false. Type compatibility is typically evaluated at run time.
- The Is operator only considers reference conversions, boxing conversions, and unboxing conversions. Other conversions, such as user-defined conversions, are not considered.

## 7.2 Collections and Iterators

In the Revit API, the Collections namespace mainly contains Array, Map, Set collections and the relevant Iterators. API Collections and Iterators are generic and type safe. For example, ElementSet and ElementIterator always contain Element and can be used as follows:

```
// elems is ElementSet
foreach(Autodesk.Revit.Element elem in elems)
{
   //… some operations
}
// elemItor is ElementIterator
elemItor.Reset();     //It is not necessary, but the proposal to do so
while(elemItor.MoveNext())
{
   Autodesk.Revit.Element elem = elemItor.Current;
   // you needn't check null for elem
}
```

All collections implement the IEnumerable interface and all relevant iterators implement the IEnumerator interface. As a result, all methods and properties are implemented in the Revit API and can play a role in the relevant collections.

The following table compares the Array, Map, and Set methods and properties.

| Function | Array | Map | Set |
|---|---|---|---|
| Add the item to the end of the collection. | Append(Object) | | |
| Removes every item from the collection rendering it empty. | Clear() | Clear() | Clear() |
| Tests for the existence of a key within the collection | | Contains(Object) | Contains(Object) |
| Removes an object with the specified key from the collection | | Erase(Object) | Erase(Object) |
| Retrieve a forward moving iterator to the collection. | ForwardIterator() | ForwardIterator() | ForwardIterator() |
| Retrieve a forward moving iterator to the collection. | GetEnumerator() | GetEnumerator() | GetEnumerator() |
| Insert the specified item into the collection. | Insert(Object, Int32) | Insert(Object, Object) | Insert(Object) |
| Retrieve a backward moving iterator to the collection. | ReverseIterator() | ReverseIterator() | ReverseIterator() |
| Test to see if the collection is empty. | IsEmpty{get;} | IsEmpty{get;} | IsEmpty{get;} |
| Gets or sets an item at a specified index (key) within the array (map) | Item(Int32) {get; set;} | Item(Object) {get; set;} | |
| Returns the number of objects in the collection. | Size{get;} | Size{get;} | Size{get;} |

**Table 13: Collections Methods and Properties**

**Note:** The GetEnumerator method uses the ForwardIterator method in its core code. Using the ForwardIterator method to get the corresponding Iterator works the same way as using GetEnumerator.

In the Revit API, Collections have their own Iterators that implement the IEnumerator interface. Use the GetEnumerator method (or ForwardIterator) to get the Iterator. To implement the IEnumerator interface, the Iterator requires the following:

- MoveNext method
- Reset method
- The Current property

| Function | Array | Map | Set |
|---|---|---|---|
| Move the iterator one item forward. | MoveNext() | MoveNext() | MoveNext() |
| Bring the iterator back to the start of the array. | Reset() | Reset() | Reset() |
| Retrieve the current focus of the iterator. | Current{get;} | Current{get;} | Current{get;} |

**Table 14: Method and Property Iterators**

In general, Array, Map, and Set are not used generically. Instead, use a collection that has an idiographic type, such as ElementSet.

Implementing all of the collections is similar. The following example uses ElementSet and ModelCurveArray to demonstrate how to use the main collection properties:

**Code Region 1-Collection**

```
Autodesk.Revit.Application app = commandData.Application;
Autodesk.Revit.Document doc = app.ActiveDocument;
Autodesk.Revit.SelElementSet selection = doc.Selection.Elements;
// Store the ModelLine(ModelArc, ModelNurbSpline) references
ModelCurveArray lineArray = new ModelCurveArray();
ModelCurveArray arcArray = new ModelCurveArray();
ModelCurveArray nurbArray = new ModelCurveArray();

// … Store operation
ElementId id = new ElementId();
id.Value = 131943;  //assume 131943 is a model line element id
lineArray.Append(doc.get_Element(ref id) as ModelLine);

// use Size property of Array
MessageBox.Show("Before Insert: " + lineArray.Size + " in lineArray," +
  arcArray.Size + " in arcArray," + nurbArray.Size + " in nurbArray.");

   // use IsEmpty property of Array
if (!lineArray.IsEmpty)
{
   // use Item(int) property of Array
   ModelCurve modelCurve = lineArray.get_Item(0) as ModelCurve;

  // erase the specific element from the set of elements
   selection.Erase(modelCurve);

   // create a new model line and insert to array of model line
   SketchPlane sketchPlane = modelCurve.SketchPlane;

   XYZ startPoint = new XYZ(0, 0, 0);  // the start point of the line
   XYZ endPoint = new XYZ(10, 10, 0);  // the end point of the line
   // create geometry line
    Line geometryLine = app.Create.NewLine(ref startPoint, ref endPoint, true);

   // create the ModelLine
    ModelLine line = doc.Create.NewModelCurve(geometryLine, sketchPlane) as
  ModelLine;

   lineArray.Insert(line, lineArray.Size - 1);
}

MessageBox.Show("After Insert: " + lineArray.Size + " in lineArray.");
```

```
// use Clear method to remove all elements in lineArray
lineArray.Clear();

MessageBox.Show("After Clear: " + lineArray.Size + " in lineArray.");
```

# 8 Editing Elements

In Revit, you can move, rotate, delete, mirror, group, and array one element or a set of elements with the Revit API. Using the editing functionality in the API is similar to the commands in the Revit UI.

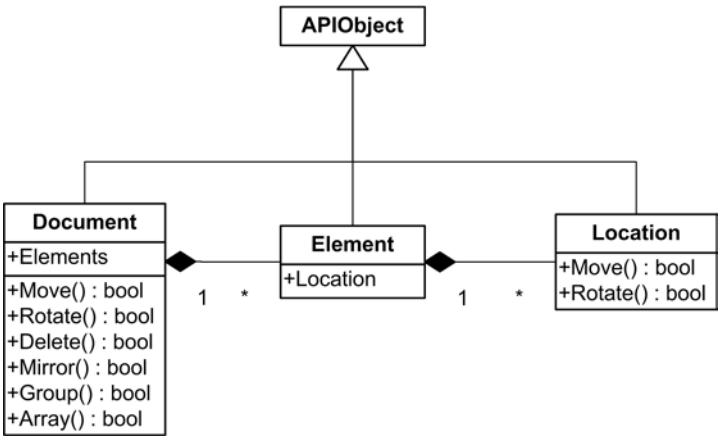



**Figure 55: Editing Elements diagram**

## 8.1 Move

The Revit API provides Move methods to move one or more elements from one place to another. It is as precise as the Move Command in the Revit UI.

| Member | Description |
|---|---|
| Move(Element, XYZ) | Move an element in the project by a specified vector. |
| Move(ElementId, XYZ) | Move an element by ID in the project by a specified vector. |
| Move(ElementIdSet, XYZ) | Move several elements by a set of IDs in the project by a specified vector. |
| Move(ElementSet, XYZ) | Move several elements in the project by a specified vector. |

**Table 15: Move Members**

**Note:** When you use the Move method, the following rules apply.

- The Move method cannot move a level-based element up or down from the level. When the element is level-based, you cannot change the Z coordinate value. However, you can place the element at any location in the same level.

  For example, if you create a new column at the original location (0, 0, 0) in Level1, and then move it to the new location (10, 20, 30), the column is placed at the location (10, 20, 0) instead of (10, 20, 30).

**Code Region 1-Editing Elements**

```
Document doc = commandData.Application.ActiveDocument;
// Get the revit document create handle
Autodesk.Revit.Creation.Document creater = doc.Create;
// Create a column with given location, column type, and level
XYZ origin = new XYZ(0, 0, 0);
Level level = GetLevel(doc);
```

```
FamilySymbol columnType = GetColumnType(doc);
FamilyInstance column = creater.NewFamilyInstance(ref origin, columnType, level,
   Autodesk.Revit.Structural.Enums.StructuralType.Column);
// Move the column to new location.
XYZ newPlace = new XYZ(10, 20, 30);
doc.Move(column, ref newPlace);
```

- When you move one or more elements, associated elements are moved. For example, if a wall with windows is moved, the windows are also moved. It is the same when moving several elements.
- If you pin the element in Revit, the Element Pinned property is true. This means that the element cannot be moved or rotated.

Another way to move an element in Revit is to use Location and its derivative objects. In the Revit API, the Location object provides the ability to translate and rotate elements. More location information and control is available using the Location object derivatives such as LocationPoint or LocationCurve. If the Location element is downcast to a LocationCurve object or a LocationPoint object, move the curve or the point to a new place directly.

**Code Region 2-Editing Elements**

```
Wall wall = GetWall(doc);
LocationCurve wallLine = wall.Location as LocationCurve;
XYZ translationVec = new XYZ(10, 20, 0);
wallLine.Move(ref translationVec);
```

When you move the element, note that the vector (10, 20, 0) is not the destination but the offset. The following picture illustrates the wall position before and after moving.

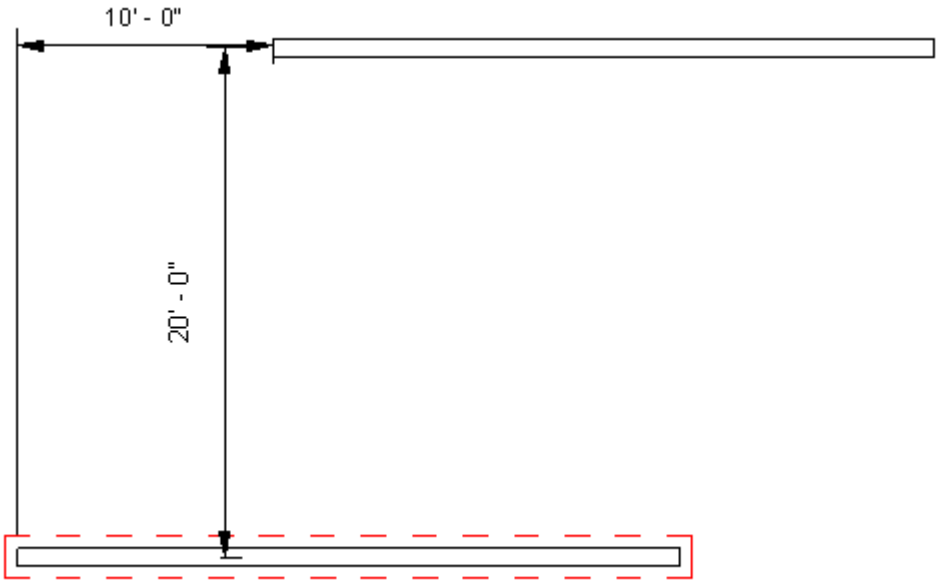



**Figure 56: Move a wall using the LocationCurve**

In addition, you can use the LocationCurve Curve property or the LocationPoint Point property to move one element in Revit.

Use the Curve property to move a curve-driven element to any specified position. Many elements are curve-driven, such as walls, beams, and braces. Also use the property to resize the length of the element.

**Code Region 3-Editing Elements**

```
//Wall wall = GetWall();
LocationCurve wallLine = wall.Location as LocationCurve;
XYZ p1 = XYZ.Zero;
XYZ p2 = new XYZ(10, 20, 0);
Line newWallLine = doc.Create.NewLineBound(ref p1, ref p2);
// Change the wall line to a new line.
wallLine.Curve = newWallLine;
```

Use the LocationPoint Point property to set the element's physical location.

**Code Region 4-Editing Elements**

```
FamilyInstance column = GetColumn(doc);
LocationPoint columnPoint = column.Location as LocationPoint;
XYZ newLocation = new XYZ(10, 20, 0);
// Move the column to the new location
columnPoint.Point = newLocation;
```

## 8.2 Rotate

The Revit API uses the Rotate methods to rotate one or several elements in the project.

| Member | Description |
|---|---|
| Rotate(Element, Line, Double) | Rotate an element in the project by a specified degree around a given axis. |
| Rotate(ElementId, Line, Double) | Rotate an element by ID in the project by a specified degree around a given axis. |
| Rotate(ElementIdSet, Line, Double) | Rotate several elements by IDs in the project by a specified degree around a given axis. |
| Rotate(ElementSet, Line, Double) | Rotate several elements in the project by a specified degree around a given axis. |

**Table 16: Rotate Members**

In the rotate methods, the angle of rotation is in radians. The positive radian means rotating counterclockwise around the specified axis, while the negative radian means clockwise, as the following pictures illustrates.

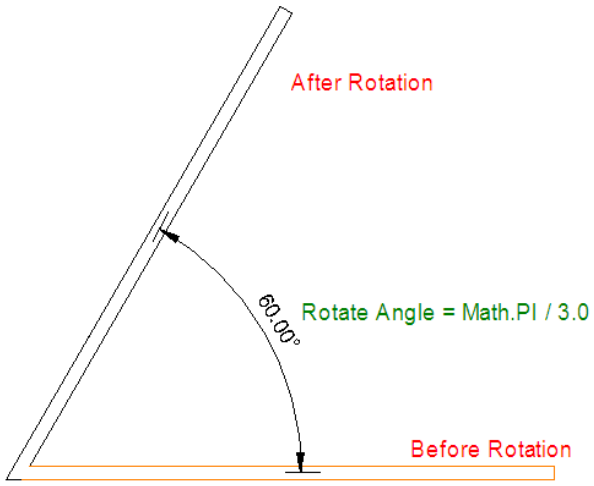



**Figure 57: Counterclockwise rotation**

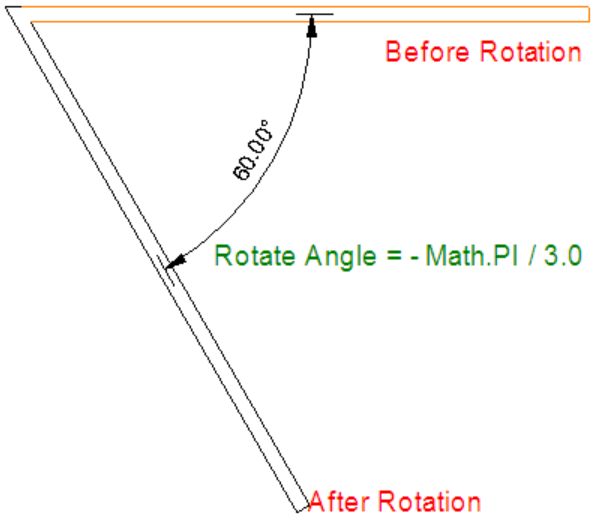



**Figure 58: Clockwise rotation**

**Note:** When you rotate an element, the axis line must be bound. An unbounded axis line results in a failed rotation even if Succeeded is returned to Revit. In this situation, the Rotate method still returns a true value.

**Code Region 5-Editing Elements**

```
Document doc = commandData.Application.ActiveDocument;
XYZ point1 = new XYZ(10, 20, 0);
XYZ point2 = new XYZ(10, 20, 30);
// The axis should be a bound line.
Line axis = commandData.Application.Create.NewLineBound(ref point1, ref point2);
bool successful = doc.Rotate(element, axis, Math.PI / 3.0);
```

If the element Location can be downcast to a LocationCurve or a LocationPoint, you can rotate the curve or the point directly.

**Code Region 6-Editing Elements**

```
XYZ aa = XYZ.Zero;
XYZ cc = XYZ.Zero;
// Rotate the element via its location curve.
LocationCurve curve = element.Location as LocationCurve;
if (null != curve)
{
    Curve line = curve.Curve;
    aa = line.get_EndPoint(0);
    cc = new XYZ(aa.X, aa.Y, aa.Z + 10);
    Line axis = commandData.Application.Create.NewLineBound(ref aa, ref cc);
    curve.Rotate(axis, Math.PI / 2.0);
}
// Rotate the element via its location point.
LocationPoint point = element.Location as LocationPoint;
if (null != point)
{
    aa = point.Point;
    cc = new XYZ(aa.X, aa.Y, aa.Z + 10);
    Line axis = commandData.Application.Create.NewLineBound(ref aa, ref cc);
    point.Rotate(axis, Math.PI / 2.0);
}
```

## 8.3 Delete

The Revit API provides Delete methods to delete one or more elements in the project.

| Member | Description |
|---|---|
| Delete(Element) | Delete an element from the project. |
| Delete(ElementId) | Delete an element from the project using the element ID |
| Delete(ElementIdSet) | Delete several elements from the project by their IDs. |
| Delete(ElementSet) | Delete several elements from the project. |

**Table 17: Delete Members**

You can delete the element specified using the element object or the ElementId. These methods delete a specific element and any elements dependent on it.

**Code Region 7-Editing Elements**

```
Document doc = commandData.Application.ActiveDocument;
// Delete a selected element.
ElementIdSet deletedIdSet1 = doc.Delete(element);
```

**Code Region 8-Editing Elements**

```
Document doc = commandData.Application.ActiveDocument;
// Delete an element via its id
ElementId elementId = element.Id;
ElementIdSet deletedIdSet2 = doc.Delete(ref elementId);
```

The API also provides two ways to delete several elements.

**Code Region 9-Editing Elements**

```
Document doc = commandData.Application.ActiveDocument;
// Delete all the selected elements via the set of elements
ElementSet elementSet = doc.Selection.Elements;
ElementIdSet deletedIdSet3 = doc.Delete(elementSet);
```

**Code Region 10-Editing Elements**

```
Document doc = commandData.Application.ActiveDocument;
// Delete all the selected elements via the set of element ids.
ElementIdSet idSelection = new ElementIdSet();
foreach (Autodesk.Revit.Element elem in doc.Selection.Elements)
{
    ElementId id = elem.Id;
    idSelection.Insert(ref id);
}
ElementIdSet deletedIdSet4 = doc.Delete(idSelection);
```

**Note:** After you delete the elements, any references to the deleted elements become invalid and throw an exception if they are accessed.

Some elements have special deletion methods. For example, the Materials.Remove method is specifically for deleting Material objects. For more information, see the Material chapter Material Management section.

## 8.4 Mirror

The Revit API uses the Mirror method to mirror one or more elements in the project.

| Member | Description |
|---|---|
| Mirror(Element, Reference) | Mirror one element by a reference. |
| Mirror(ElementId, Reference) | Mirror one element by a reference and the element ID. |
| Mirror(ElementIdSet, Reference) | Mirror several elements by a reference and their IDs. |
| Mirror(ElementSet, Reference) | Mirror several elements by a reference. |

**Table 18: Mirror Members**

There are two ways to mirror one element and two ways to mirror several elements. After performing the mirror operation, you can access the new elements from the Selection ElementSet. The following code illustrates how to mirror a column using its reference line, then move the new column to a new location.

**Note:** This code only works in Revit Structure.

**Code Region 11-Editing Elements**

```
Document doc = commandData.Application.ActiveDocument;
Selection choices = doc.Selection;
ElementSet collection = choices.Elements;
Autodesk.Revit.Element currentComponent = null;
foreach (Autodesk.Revit.Element elem in collection)
{
   currentComponent = elem;
}
FamilyInstance column = currentComponent as FamilyInstance;
if (null != column)
{
   AnalyticalModelFrame analytical = column.AnalyticalModel as AnalyticalModelFrame;
   Reference lineReference = analytical.Curve.Reference;
   doc.Mirror(currentComponent, lineReference);
   //The selected element is changed after the Mirror operation.
   foreach (Autodesk.Revit.Elements.FamilyInstance newColumn in choices.Elements)
   {
       // true signifies the column is new.
       bool isMirror = newColumn.Mirrored;

       //Offset the new column
       XYZ translationVec = new XYZ(10, 20, 30);
       newColumn.Location.Move(ref translationVec);
   }
}
```

Every FamilyInstance has a Mirrored property. It indicates whether a FamilyInstance (for example a column) is mirrored. In the previous example, if the column Mirrored property returns true, then you have a mirrored column.

**Note:** In Revit, the Mirror command mirrors a selected modeling component. However, in the Revit API, if you get a Geometry.Line from ModelLine.Geometry, its Reference returns null. You must get a plane or line Reference from model element. For this reason, the Mirror method is not easy to use.

## 8.5 Group

The Revit API uses the Creation.Document.NewGroup method to select an element or multiple elements or groups then combines them, so that they can be modified as one large element.

**Code Region 12-Editing Elements**

```
Document doc = commandData.Application.ActiveDocument;
Selection choices = doc.Selection;
ElementSet collection = choices.Elements;
// Group all selected elements
Group group = doc.Create.NewGroup(collection);
```

Initially, the group has a generic name, such as Group 1. It can be modified by changing the name of the group type as follows:

**Code Region 13-Editing Elements**

```
// Change the default group name to a new name “MyGroup”
group.GroupType.Name = "MyGroup";
```

Like the Revit UI, there are three types of groups in the Revit API; Model Group, Detail Group, and Attached Detail Group. All are created using the NewGroup method. The created Group’s type depends on the Elements passed.

- If no detail Element is passed, Model Group is created.
- If all Elements are detail, the Detail Group is created.
- If both Elements are included, a Model Group that contains an Attached Detail Group is created and returned.

**Note:** When elements are grouped, they can be deleted from the project.

- When a model element in a model group is deleted, it is still visible when the mouse cursor hovers over or clicks the group, even if the application returns Succeeded to the UI. In fact, the model element is deleted and you cannot select or access that element.
- When the last member of a group instance is deleted, excluded, or removed from the project, the model group instance is deleted.
- When a detail element in a detail group or an attached group is excluded, the detail element is invisible. However, you can use the Place Detail button on the Options bar to place the attached detail group to the selected model group again. The following figure illustrates the Attached Detail Group Placement dialog box.

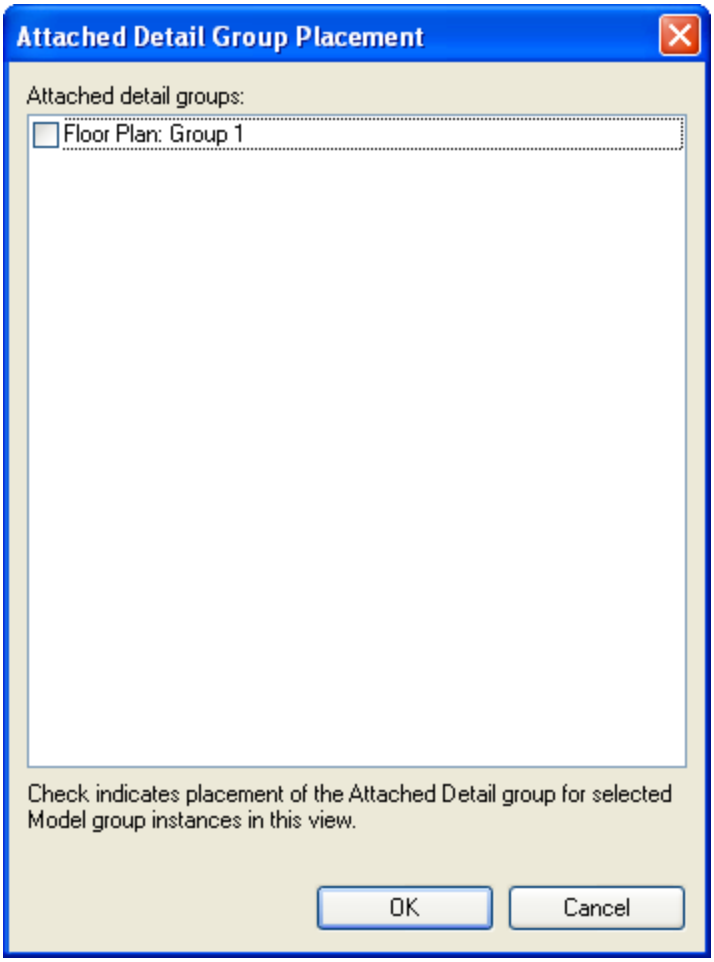

Attached Detail Group Placement
Attached detail groups:
Floor Plan: Group 1
Check indicates placement of the Attached Detail group for selected Model group instances in this view.
OK
Cancel

**Figure 59: Attached detail group placement**

When elements are grouped, they cannot be moved or rotated. If you perform these operations on the grouped elements, nothing happens to the elements though the Move or Rotate method returns true.

You cannot group dimensions and tags without grouping the elements they reference. If you do, the Revit API fails and throws the following exception.

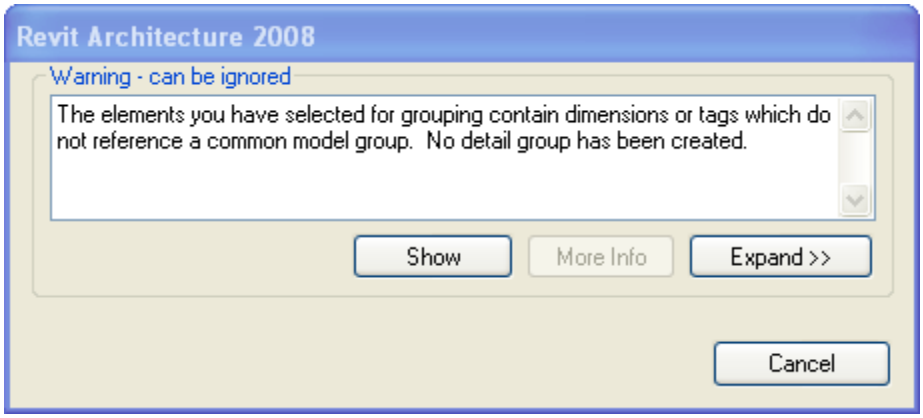

Revit Architecture 2008
Warning - can be ignored
The elements you have selected for grouping contain dimensions or tags which do not reference a common model group. No detail group has been created.
Show
More Info
Expand >>
Cancel

**Figure 60: Grouping dimension exception without a reference**

You can group dimensions and tags that refer to model elements in a model group. The dimensions and tags are added to an attached detail group. The attached detail group cannot be moved, copied, rotated, arrayed, or mirrored without doing the same to the parent group.

## 8.6 Array

The Revit API uses the Array methods to array one or more elements in the project. These methods create a linear or radial array of one or more selected components. For example, you can select a door and windows located in the same wall then create multiple instances of the door, wall, and window configuration.

| Member | Description | Array Type |
|---|---|---|
| Array(Element, Int32, Boolean, XYZ) | Array one or several elements in the project by a specified number. | Linear |
| Array(Element, Int32, Boolean, Double) | Array one or several elements in the project by a specified degree. | Radial |
| Array(ElementId, Int32, Boolean, XYZ) | Array one or several elements in the project by a specified number. | Linear |
| Array(ElementId, Int32, Boolean, Double) | Array one or several elements in the project by a specified degree. | Radial |
| Array(ElementIdSet, Int32, Boolean, XYZ) | Array one or several elements in the project by a specified number. | Linear |
| Array(ElementIdSet, Int32, Boolean, Double) | Array one or several elements in the project by a specified degree. | Radial |
| Array(ElementSet, Int32, Boolean, XYZ) | Array one or several elements in the project by a specified number. | Linear |
| Array(ElementSet, Int32, Boolean, Double) | Array one or several elements in the project by a specified degree. | Radial |

**Table 19 : Document Array Members**

There are two types of arrays: linear array and radial array.

- Linear arrays represent an array created from one point to another point in the Revit project.
- Radial arrays represent an array created around a central point in the Revit project.

The Array method is useful if you need to create several instances of a component and manipulate them simultaneously. Every instance in an array can be a member of a group.

**Note:** When using the Array method, the following rules apply:

- When performing Linear and Radial Array operations, elements dependent on the arrayed elements are also arrayed.
- Some elements cannot be arrayed because they cannot be grouped.
- Arrays are not supported by most annotation symbols.

The Revit API also provides ArrayWithoutAssociate methods to array one or several elements without being grouped and associated. This method is similar to the Array method.

## 8.7 SuspendUpdating

SuspendUpdating is a class that temporarily delays automatic updating and consistency operations in Revit. For example, if you create an instance of this object, it suspends certain updating and consistency operations to increase large-scale change performance using the API.

Updating continues when the SuspendUpdating object is explicitly disposed. Currently, the SuspendUpdating function works primarily with object transformation operations such as move and rotate.

**Code Region 14-Editing Elements**

```
Document doc = commandData.Application.ActiveDocument;
Selection choices = doc.Selection;
ElementSet collection = choices.Elements;
// Using statement defines a scope at the end of which the suspandUpdating
// object is disposed.
using (SuspendUpdating aSuspendUpdating = new SuspendUpdating(doc))
{
   // Suspend these operations.
   foreach (FamilyInstance element in collection)
   {
      MoveRotateElement(commandData.Application.Create, doc, element);
   }
}
// Update operations when the SuspendUpdating object is disposed.
```

SuspendUpdating objects can be nested. The Nested property returns true if the object is already nested. During nesting, the final update happens only when all SuspendUpdating objects are disposed. For example, if the following function is called by the previous example, the bSuspendUpdating object Nested property returns true. In this situation, the Move operation update only happens after both bSuspendUpdating and aSuspendUpdating objects are disposed.

**Note:** The aSuspendUpdating object Nested property still returns false. While this object is active, Revit model data can be inconsistent.

**Code Region 15-Editing Elements**

```
private bool MoveRotateElement(Autodesk.Revit.Creation.Application              appCreation,
   Document doc, Autodesk.Revit.Element elem)
{
  using (SuspendUpdating bSuspendUpdating = new SuspendUpdating(doc))
  {
    // Move Element
     XYZ translation3D = new XYZ(0.0, 0.0, 0.0);
     bool successful1 = doc.Move(elem, ref translation3D);

     // Rotate Element
     XYZ point1 = new XYZ(10, 20, 0);
     XYZ point2 = new XYZ(10, 20, 30);
     Line axis = appCreation.NewLineUnbound(ref point1, ref point2);
     bool successful2 = doc.Rotate(elem, axis, Math.PI / 3.0);

     return (successful1 && successful2);
   }
}
```

# 9 Host Elements and Opening

This chapter discusses Elements and the corresponding Symbols representing built-in place construction:

- HostObject - The first two sections focus on HostObject and corresponding HostObjAttributes subclasses shown in the following diagram:

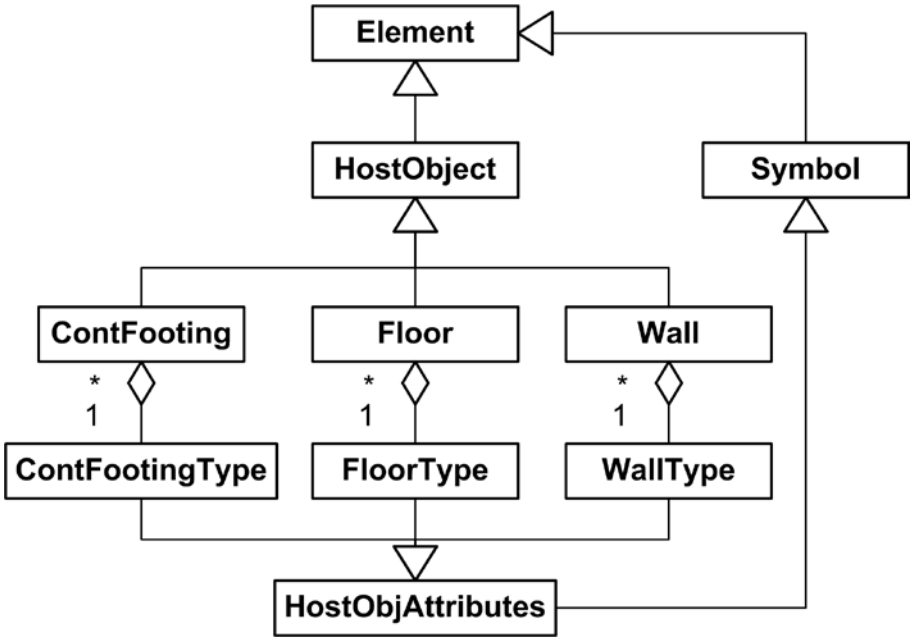



**Figure 61: HostObject and HostObjAttributes diagram**

- Foundation - Different foundations in the API are represented as different classes, including Floor, ContFooting, and FamilyInstance. The Floor and Foundation section compares them in the API.
- CompoundStructure - This section describes the HostObject.CompoundStructure property and provides tips to access Material.

In addition to host Elements, the Opening class is introduced at the end of this chapter.

## 9.1 Wall

There are four kinds of Walls represented by the WallType.WallKind enumeration:

- Stacked
- Curtain
- Basic
- Unknown

The Wall and WallType class work with the Basic wall type while providing limited function to the Stacked and Curtain walls. On occasion you need to check a Wall to determine the wall type. For example, you cannot get sub-walls from a Stacked Wall or get mullions from a Curtain Wall using the API. WallKind is read only and set by System Family.

The Wall.Flipped property and Wall.flip method gain access to and control Wall Orientation. In the following examples, a Wall is compared before and after calling the flip method.

- The Orientation property before is (0.0, 1.0, 0.0).

- The Orientetion property after the flip call is (0.0, -1.0, 0.0).
- The Wall Location Line (WALL_KEY_REF_PARAM) parameter is 3, which represent the Finish Face: Interior in the following table.
- Taking the line as reference, the Wall is moved but the Location is not changed.

**Figure 62: Original wall**

**Figure 63: Wall after flip**

| Location Line Value | Description |
|---|---|
| 0 | Wall Centerline |
| 1 | Core Centerline |
| 2 | Finish Face: Exterior |
| 3 | Finish Face: Interior |
| 4 | Core Face: Exterior |
| 5 | Core Face: Interior |

**Table 20: Wall Location Line**

There are five override methods to create a Wall:

```
public Wall NewWall(Curve curve, Level level, bool structural);
```

```
public Wall NewWall(Curve curve, WallType wallType, Level level, double height, double
   offset,
   bool flip, bool structural);
```

```
public Wall NewWall(CurveArray profile, WallType wallType, Level level, bool structural);
```

```
public Wall NewWall(CurveArray profile, WallType wallType, Level level, bool structural,
   ref XYZ normal);
```

```
public Wall NewWall(CurveArray profile, bool structural);
```

The first two methods create rectangular Walls and the remaining three methods create Walls with vertical profiles that are not rectangular. In the Revit API, Stacked wall and Curtain wall can be a non-rectangular shape.

The WallType Wall Function (WALL_ATTR_EXTERIOR) parameter influences the created wall instance Room Bounding and Structural Usage parameter. The WALL_ATTR_EXTERIOR value is an integer:

| Wall Function | Interior | Exterior | Foundation | Retaining | Soffit |
|---|---|---|---|---|---|
| Value | 0 | 1 | 2 | 3 | 4 |

**Table 21: Wall Function**

The following rules apply to Wall:

- If the input structural parameter is true or the Wall Function (WALL_ATTR_EXTERIOR) parameter is Foundation, the Wall StructuralUsage parameter is Bearing; otherwise it is NonBearing.
- The created Wall Room Bounding (WALL_ATTR_ROOM_BOUNDING) parameter is false if the Wall Function (WALL_ATTR_EXTERIOR) parameter is Retaining.
- Wall.WallType is preferred to Wall.ObjectType because it has a setter and does not need to be downcast to WallType.

For more information about structure related functions such as the AnalyticalModel property, refer to the Revit Structure chapter.

## 9.2 Floor and Foundation

Floors and Foundations are located in the Design bar:

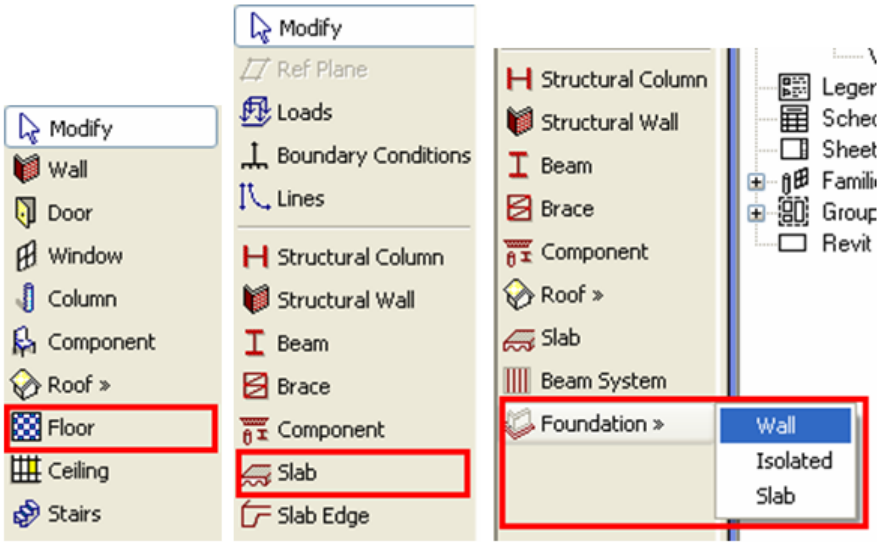



**Figure 64: Floors and foundations**

Their counterparts in the API include:

| Object | Element Type | Symbol Type | Element Creation | Other |
|---|---|---|---|---|
| Floor | Floor | FloorType | NewFloor/ NewSlab | StructuralUsage = InstanceUsage.Slab<br>FloorType.IsFoundationSlab = false |

| Object | Element Type | Symbol Type | Element Creation | Other |
|---|---|---|---|---|
| Slab | Floor | FloorType | NewSlab | StructuralUsage = InstanceUsage.Slab<br>FloorType.IsFoundationSlab = false |
| Wall Foundation | ContFooting | ContFootingType | No | Category = OST_StructuralFoundation |
| Isolated Foundation | FamilyInstance | FamilySymbol | NewFamilyInstance | Category = OST_StructuralFoundation |
| Foundation Slab | Floor | FloorType | NewFloor | Category = OST_StructuralFoundation<br>StructuralUsage = InstanceUsage.SlabFoundation<br>FloorType.IsFoundationSlab = true |

**Table 22: Floors and Foundations in the API**

The following rules apply to Floor:

- Elements created from the Foundation Design bar have the same category, OST_StructuralFoundation, but correspond to different Classes.
- The FloorType IsFoundationSlab property sets the FloorType category to OST_StructuralFoundation or not.

When you retrieve FloorType to create a Floor or Foundation Slab with NewFloor, use the following method:

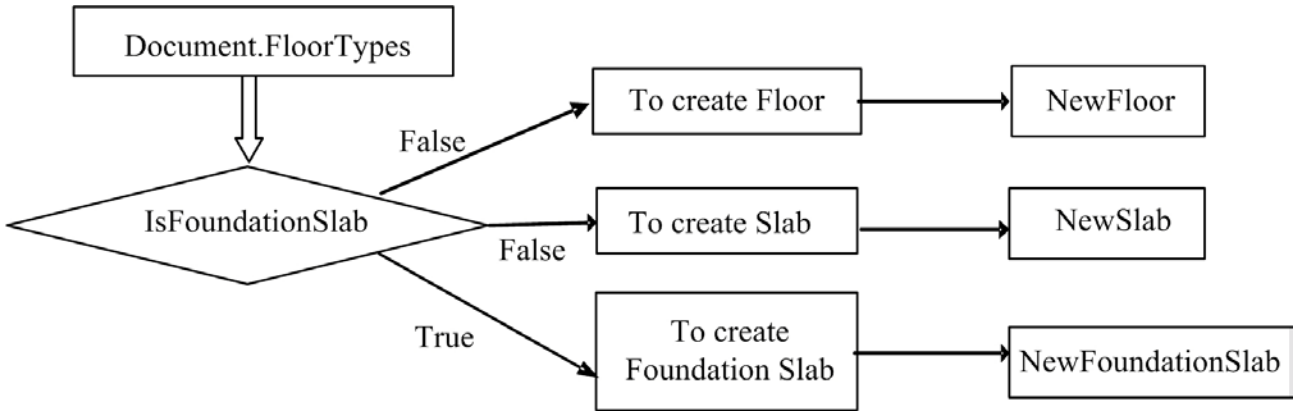



**Figure 65: Create foundation and floor / slab**

NewSlab is an override NewFloor method. Currently, the API does not provide access to the Floor Slope Arrow in the Floor class. However, in Revit Structure, you can create a sloped slab with NewSlab:

```
public Floor NewSlab(CurveArray profile, Level level, Line slopedArrow, double angle,
   bool isImperial, bool isStructural);
```

Slope Arrow is created using the slopedArrow parameter.

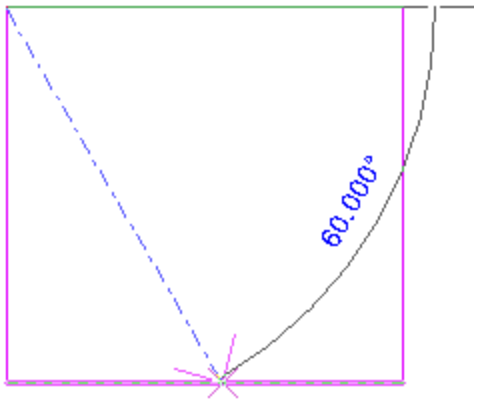



**Figure 66: slopedArrow parameter in NewSlab**

Floor.FloorType is preferred to Floor.ObjectType because it has a setter and does not need to be downcast to FloorType. For more information about structure related functions such as SpanDirectionSymbols and SpanDirectionAngle, refer to the Revit Structure chapter.

When editing an Isolated Foundation in Revit, you can perform the following actions:

- You can pick a host, such as a floor. However, the FamilyInstance object Host property always returns null.
- When deleting the host floor, the Foundation is not deleted with it.
- The Foundation host is available from the Host (INSTANCE_FREE_HOST_PARAM) parameter.
- Use another related Offset (INSTANCE_FREE_HOST_OFFSET_PARAM) parameter to control the foundation offset from the host Element.

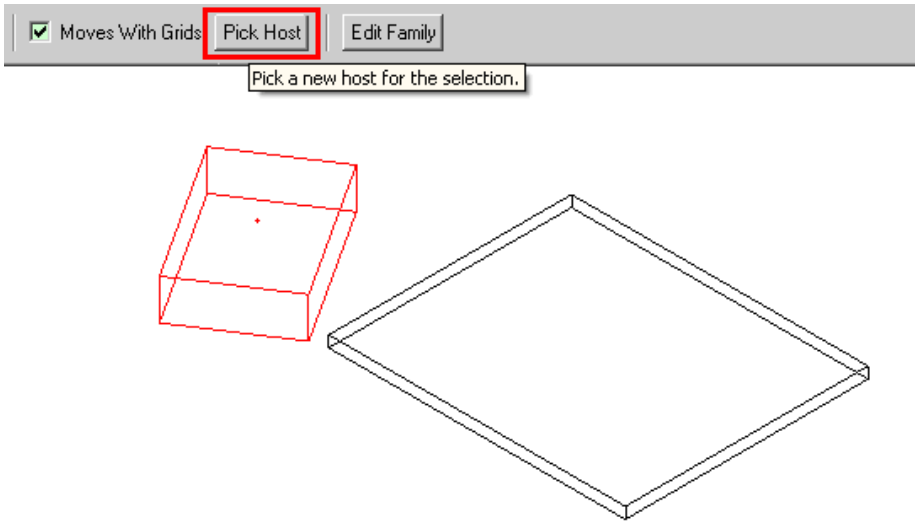



**Figure 67: Pick Host for FoundationSlab (FamilyInstance)**

Continuous footings are represented by the ContFooting class in the API. Currently, the API provides limited access to both ContFooting and ContFootingType except when using AnalyticalModel (refer to the AnalyticalModel section in the Revit Structure chapter). For example, the attached wall is not available in Revit Architecture. In Revit Structure, the relationship between the Wall class and the ContFooting class is shown using SupportData in the structural Wall class AnalyticalModelWall. For more details, refer to the SupportData section in the Revit Structure chapter.

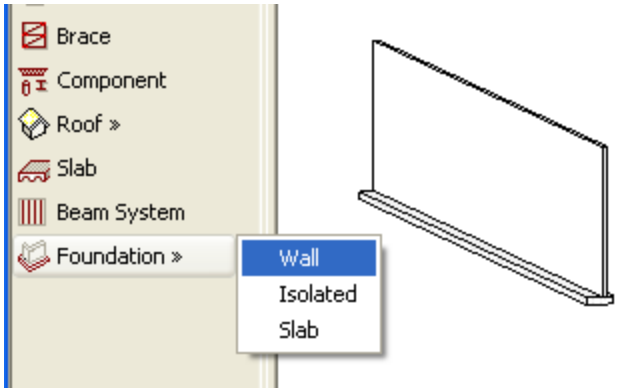



**Figure 68: Wall ContFooting**

## 9.3 Other Elements

The following Elements are part of Host Elements but they exist as special cases.

### 9.3.1 Stair and Ramp

Stair, Ramp, and the associated symbols do not have specific classes in the API.

- Stair and its associated symbol are represented as Element and Symbol in the OST_Stairs category.
- Ramp and its symbol are represented as Element and Symbol in the OST_Ramps category.

### 9.3.2 Ceiling and Roof

Ceiling does not have a specific class. (The CeilingAndFloor class is not its corresponding class.)

- The Ceiling object is an Element in the OST_Ceilings category.
- The Ceiling symbol is HostObjAttributes using the OST_Ceilings category with its CompoundStructure available.

Roof does not have a specific class.

- The Roof object is an Element in the OST_Roofs category.
- The Roof symbol is HostObjAttributes in the OST_Roofs category with its CompoundStructure available.

## 9.4 CompoundStructure

Some host elements, such as Walls, Floors, Ceilings, and Roofs, include parallel layers. The parallel layers are available from the HostObjAttributes.CompoundStructure property:

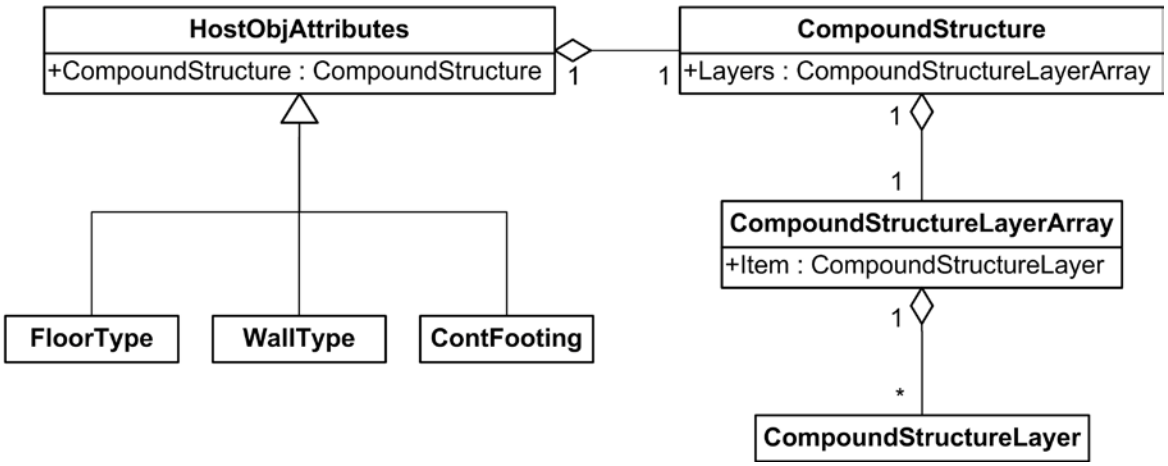



**Figure 69: ComponentStructure diagram**

**Note**: The following items are important when using CompoundStructure:

- The total thickness of the element is the sum of each CompoundStructureLayer's thickness. You cannot change the element's total thickness directly but you can change it via changing the CompoundStructureLayer thickness.
- CompoundStructure is a property of HostObjAttributes. Changing the CompoundStructureLayer changes every element instance in the current document.
- The CompoundStructureLayerArray ReadOnly property retrieved from the element CompoundStructure is always true. You cannot insert, remove, or reorder CompoundStructureLayer within the array.
- The CompoundStructureLayer DeckProfile and DeckUsage properties only work with Slab in Revit Structure. For more details, refer to the Revit Structure chapter.
- The wall-related information outlined in the rectangles in the following picture are not exposed by the API.

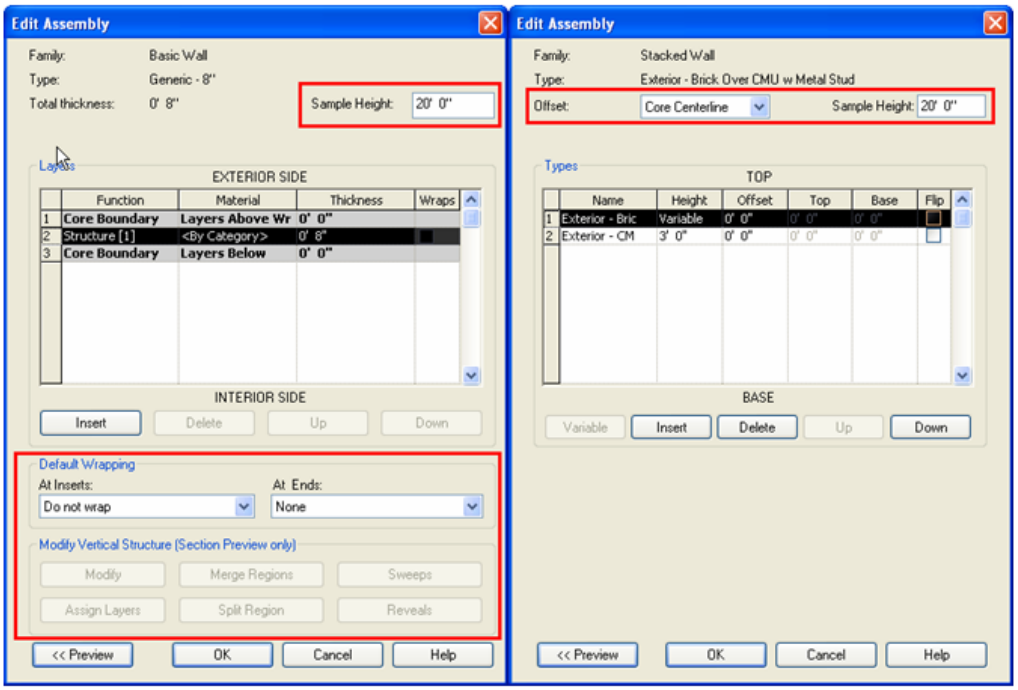

Edit Assembly
Family: Basic Wall
Type: Generic - 8"
Total thickness: 0' 8"
Sample Height: 20' 0"
Layers
EXTERIOR SIDE
Function Material Thickness Wraps
1 Core Boundary Layers Above Wr 0' 0"
2 Structure [1] <By Category> 0' 8"
3 Core Boundary Layers Below 0' 0"
INTERIOR SIDE
Insert Delete Up Down
Default Wrapping
At Inserts: Do not wrap
At Ends: None
Modify Vertical Structure (Section Preview only)
Modify Merge Regions Sweeps
Assign Layers Split Region Reveals
<< Preview OK Cancel Help
Edit Assembly
Family: Stacked Wall
Type: Exterior - Brick Over CMU w Metal Stud
Offset: Core Centerline
Sample Height: 20' 0"
Types
TOP
Name Height Offset Top Base Flip
1 Exterior - Bric Variable 0' 0"
2 Exterior - CM 3' 0" 0' 0"
BASE
Variable Insert Delete Up Down
<< Preview OK Cancel Help

**Figure 70: Non-Supported CompoundStructure functionality in the API**

### 9.4.1 Material

Each CompoundStructureLayer in HostObjAttributes is typically displayed with some type of material. If CompoundStructureLayer.Material returns null, it does not mean the layer does not have Material. Material is Category-related. For more details, refer to the Material chapter. Getting the CompoundStructureLayer Material is illustrated in the following sample code:

**Code Region 1-Host Elements and Opening**

```
// ... get selected wall
WallType aWallType = aWall.WallType;
// Only Basic Wall has compoundStructure
if (WallType.WallKind.Basic == aWallType.Kind)
{
  // Get CompoundStructure
  CompoundStructure comStruct = aWallType.CompoundStructure;
  Categories allCategories = doc.Settings.Categories;

  // Get the category OST_Walls default Material;
  // it is the category OST_Walls default Material
  // CompoundStructureLayer if that layer's default Material is
  // <By Category>
  Category wallCategory = allCategories.get_Item(BuiltInCategory.OST_Walls);
  Autodesk.Revit.Elements.Material wallMaterial = wallCategory.Material;

  foreach (CompoundStructureLayer structLayer in comStruct.Layers)
  {
     Autodesk.Revit.Elements.Material layerMaterial = structLayer.Material;
    // If CompoundStructureLayer's Material is specified, use default
    // Material of its Category
    if (null == layerMaterial)
    {
      switch (structLayer.Function)
      {
      case CompoundStructureLayerFunction.Finish1:
        layerMaterial =
          allCategories.get_Item(BuiltInCategory.OST_WallsFinish1).Material;
          break;
       case CompoundStructureLayerFunction.Finish2:
          layerMaterial =
          allCategories.get_Item(BuiltInCategory.OST_WallsFinish2).Material;
          break;
       case CompoundStructureLayerFunction.MembraneLayer:
          layerMaterial =
          allCategories.get_Item(BuiltInCategory.OST_WallsMembrane).Material;
          break;
       case CompoundStructureLayerFunction.Structure:
          layerMaterial =
```

```
            allCategories.get_Item(BuiltInCategory.OST_WallsStructure).Material;
            break;
         case CompoundStructureLayerFunction.Substrate:
            layerMaterial =
            allCategories.get_Item(BuiltInCategory.OST_WallsSubstrate).Material;
          break;
         case CompoundStructureLayerFunction.ThermalOrAir:
            layerMaterial =
           allCategories.get_Item(BuiltInCategory.OST_WallsInsulation).Material;
            break;
         default:
            // It is impossible run here
            break;
         }
         if (null == layerMaterial)
         {
          // If sub Category's Material returns null,
            // CompoundStructureLayer's default Material is its
          // SubCategory's
            layerMaterial = wallMaterial;
         }
      }
   }
}
```

## 9.5 Opening

In the Revit API, the Opening object is derived from the Element object and contains all of the Element object properties and methods. To retrieve all Openings in a project, use Document.ElementFilterIterator to find the Elements.Opening objects.

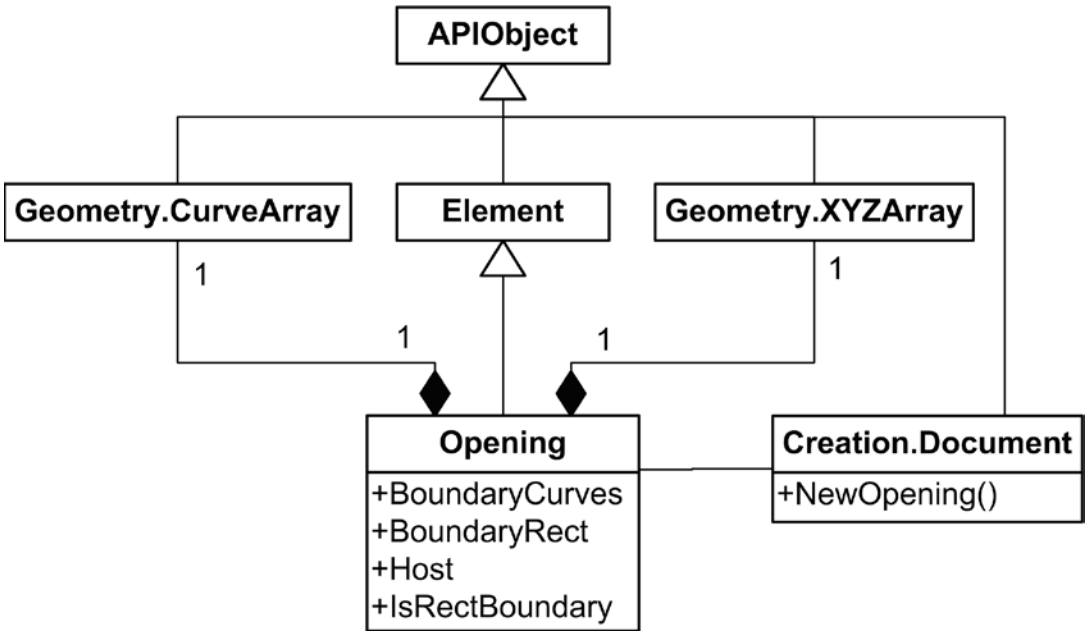



**Figure 71: Opening diagram**

### 9.5.1 General Properties

This section explains how to use the Opening properties.

- IsRectBoundary - Identifies whether the opening has a rectangular boundary.
  - If true, it means the Opening has a rectangular boundary and you can get an XYZArray object from the Opening BoundaryRect property. Otherwise, the property returns null.
  - If false, you can get a CurveArray object from the BoundaryCurves property.
- BoundaryCurves - If the opening boundary is not a rectangle, this property retrieves geometry information; otherwise it returns null. The property returns a CurveArray object containing the curves that represent the Opening object boundary.

  For more details about Curve, refer to the Geometry chapter.
- BoundaryRect - If the opening boundary is a rectangle, you can get the geometry information using this property; otherwise it returns null.
  - The property returns an XYZArray object containing the XYZ coordinates.
  - The XYZArray usually contains the rectangle boundary minimum (lower left) and the maximum (upper right) coordinates.
- Host - The host property retrieves the Opening host element. The host element is the element cut by the Opening object.

  **Note**: If the Opening object category is Shaft Openings, the Opening host is null.

The following example illustrates how to retrieve the existing Opening properties.

**Code Region 2- Host Elements and Opening**

```
private void Getinfo_Opening(Opening opening)
{
   //…
   //Get the information whether the opening has a rect boundary.
   //If the opening has a rect boundary, get the geometry
```

```
   //information from BoundaryRect property.
    //Otherwise get the geometry information from
   //BoundaryCurves property.
    if (opening.IsRectBoundary)
    {
       message += "\nThe opening has a rect boundary.";
       //An array contain two XYZ struct store the max and min
      //coordinate of rectangular
       XYZArray boundaryRect = opening.BoundaryRect;

       //Get the coordinate value of the min coordinate point
       XYZ point = opening.BoundaryRect.get_Item(0);
      message +="\nMin coordinate point: " + XYZToString(point);

       //Get the coordinate value of the Max coordinate point
       point = opening.BoundaryRect.get_Item(1);
       message += "\nMax coordinate point: " + XYZToString(point);
    }
    else
    {
       message += "\nThe opening doesn't have a rect boundary.";
       // Get curve number
       int curves = opening.BoundaryCurves.Size;
       message += "\nCurve number is : " + curves;
       for (int i = 0; i < curves; i++)
       {
          Autodesk.Revit.Geometry.Curve curve = opening.BoundaryCurves.get_Item(i);
          // Get curve start point
          message += "\nCurve start point: " + XYZToString(curve.get_EndPoint(0));
          // Get curve end point
          message += "; Curve end point: " + XYZToString(curve.get_EndPoint(1));
       }
    }
    MessageBox.Show(message, "Revit");
}
```

### 9.5.2 Create Opening

In the Revit API, use the NewOpening method to create an opening in your project. There are four method overloads you can use to create openings in different host elements:

```
//Create a new Opening in a beam, brace and column.
public Opening NewOpening(Element famInstElement, CurveArray profile, eRefFace iFace);
```

```
//Create a new Opening in a roof, floor and ceiling.
public Opening NewOpening(Element hostElement, CurveArray profile, bool bPerpendicularFace);
```

```
//Create a new Opening Element.
public Opening NewOpening(Level bottomLevel, Level topLevel, CurveArray  profile);
```

```
//Create an opening in a straight wall or arc wall.
public Opening NewOpening(Wall,ref XYZ pntStart, ref XYZ pntEnd);
```

- Create an Opening in a Beam, Brace, or Column - Use to create an opening in a family instance. The iFace parameter indicates the face on which the opening is placed.
- Create a Roof, Floor, or Ceiling Opening - Use to create an opening in a roof, floor, or ceiling.
  - The bPerpendicularFace parameter indicates whether the opening is perpendicular to the face or vertical.
  - If the parameter is true, the opening is perpendicular to the host element face. See the following picture:

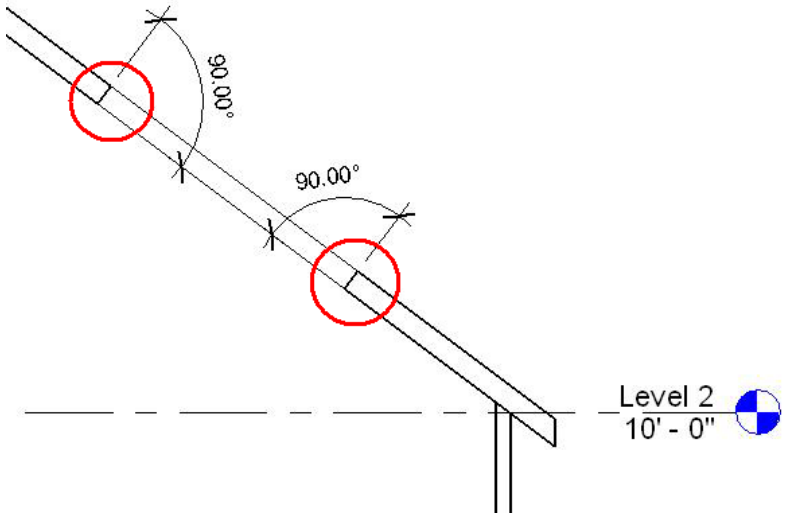



**Figure 72: Opening cut perpendicular to the host element face**

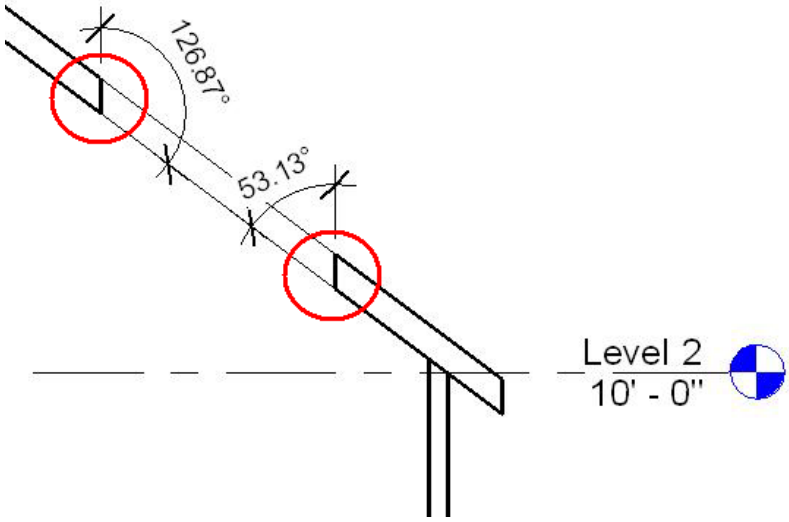



**Figure 73: Opening cut vertically to the host element**

- Create a New Opening Element - Use to create a shaft opening in your project. However, make sure the topLevel is higher than the bottomLevel; otherwise an exception is thrown.
- Create an Opening in a Straight Wall or Arc Wall - Use to create a rectangle opening in a wall. The coordinates of pntStart and pntEnd should be corner coordinates that can shape a rectangle. For example, the lower left corner and upper right corner of a rectangle. Otherwise an exception is thrown.

**Note**: Using the Opening command you can only create a rectangle shaped wall opening. To create some holes in a wall, edit the wall profile instead of the Opening command.

# 10 Component Elements

Component Elements are FamilyInstance objects that belong to the Model Element. Model Element is classified in the following way:

- Non-FamilyInstance - Elements in the System Family
- FamilyInstance – Elements in the Component Family.

In this chapter, you will learn the following:

- Relationship between family and family instance
- Family and family instance features
- How to load or create family and family instance features.

## 10.1 Identifying Elements

In Revit, the easiest way to judge whether an element is a FamilyInstance or not is by using the properties dialog box.

- If the family name starts with System Family and the Load button is disabled, it belongs to System Family.

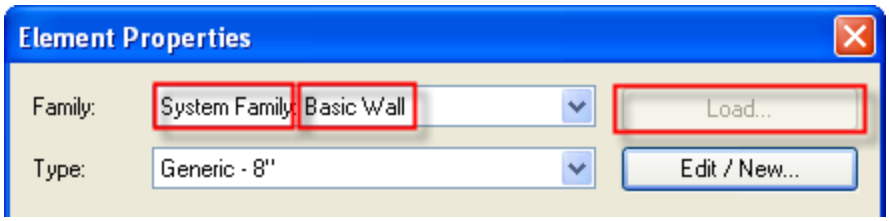



**Figure 74: System Family**

- A general FamilyInstance, which belongs to the Component Family, does not start with System Family.

  For example, in the following picture the family name for the desk furniture is Desk. In addition, the Load button is enabled.

**Figure 75: Component Family**

- The exceptions are Mass and in-place member. The Family and Type fields are blank.

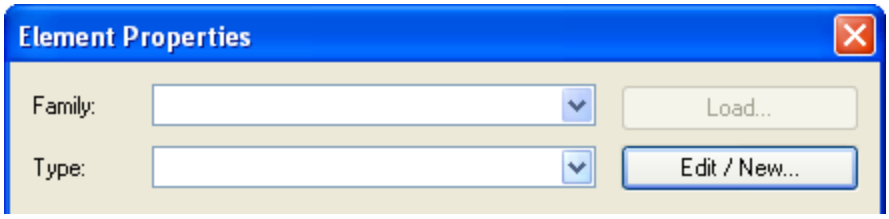



**Figure 76: System Family and Component Family exception example**

Families in the Revit API are represented by three objects:

- Family
- FamilySymbol
- FamilyInstance

Each object plays a significant role in the family structure.

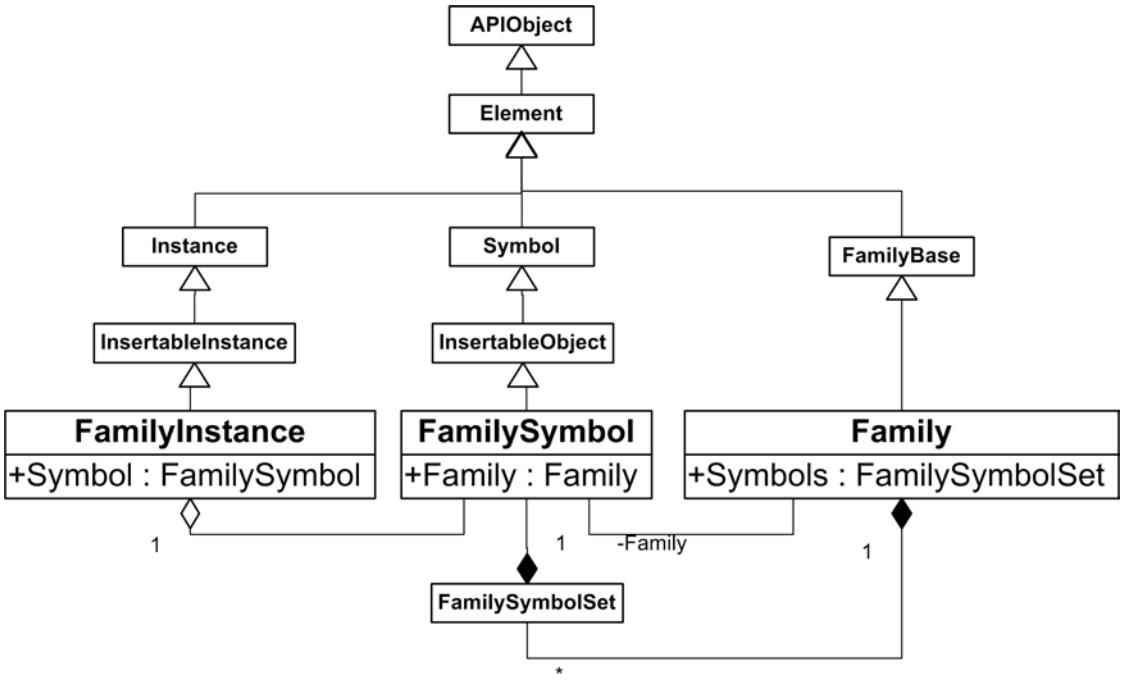



**Figure 77: Family-related class diagram**

The Family object represents an entire family such as a Single-Flush door. For example, the Single-Flush door corresponds to the Single-Flush.rfa file. The Family object contains several FamilySymbols that are used to get all family symbols to facilitate swapping instances from one symbol to another.

The FamilySymbol object represents a specific set of family settings corresponding to a Type in the Revit UI, such as 34”X80”.

The FamilyInstance object represents an actual Type (FamilySymbol) instance in the Revit project. For example，in the following picture, the FamilyInstance is a single door in the project.

- Each FamilyInstance has one FamilySymbol. The door is an instance of a 34”X80”.
- Each FamilySymbol belongs to one Family. The 34”X80” symbol belongs to a Single-Flush family.
- Each Family contains one or more FamilySymbols. The Single-Flush family contains a 34”X80” symbol, a 34”X84” symbol, a 36”X84” and so on.

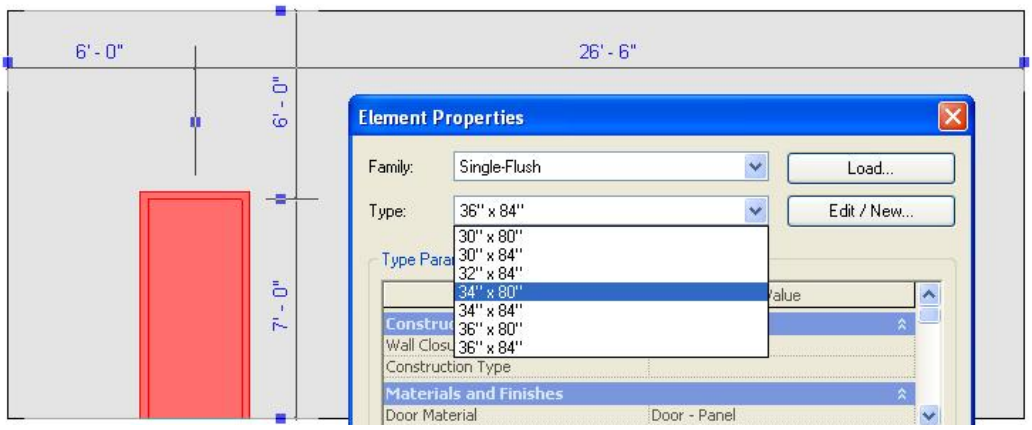



**Figure 78: Change an instance FamilySymbol**

In the previous picture, notice that you can change the FamilyInstance symbol object from one FamilySymbol object to another in the set.

It is noteworthy that while most component elements are exposed through the API classes FamilySymbol and FamilyInstance, some have been wrapped with specific API classes. For example, AnnotationSymbolType wraps FamilySymbol and AnnotationSymbol wraps FamilyInstance. More of these may be coming in the future.

## 10.2 Family

The Family class has important properties including the following:

- Components
- SolidForms
- VoidForms
- Others
- and more

### 10.2.1 Components

Components return a read-only set of all components belonging to a particular family. Components represent the auxiliary parts, instead of the main body.

```
Autodesk.Revit.ElementSet Components {get;}
```

In the following example, in which you load a table family then select the chairs individually, the chairs are family components.

1. Load a table family from the file Table-Dining Round w Chairs.rfa into the Revit project.
2. Create a family instance by adding a family member, such as 60” Diameter. A table with 6 chairs appears in the view.
3. Select the newly created table and choose Edit Family. You can select each chair individually.

Each family has a default family symbol. The family Components returns the default family symbol auxiliary components. In the table and chairs example, the family has three family symbols which have four, six and ten chairs. However the Components property returns six chairs, which is the default family symbol component.

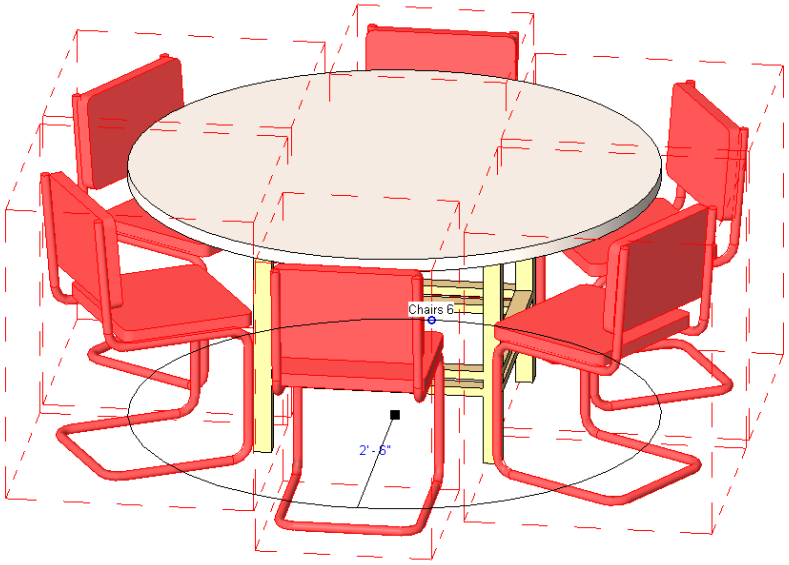



**Figure 79: Table-Dining Round w Chairs.rfa family components**

**Note**: Many Families have no Components and return and empty set. For example, the Desk family Components property returns an empty set because Desk does not have components.

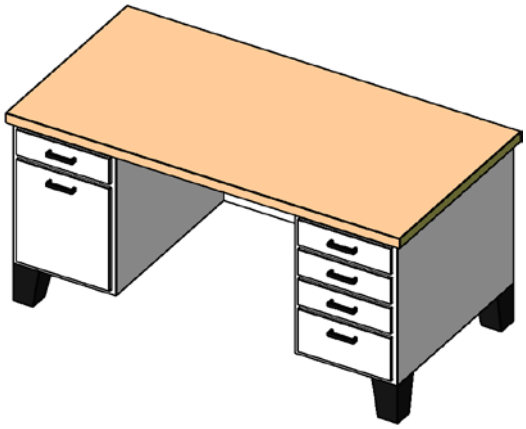


**Figure 80: A Desk has no components**

### 10.2.2 Solid and Void Forms

Both solid forms and void forms are generic form models in Revit. The void form is used to cut off a part from the solid form. Both forms have the following characteristics:

- Both forms are GenericForm class objects with a different Boolean property, IsSolid, to distinguish them.
- The GenericForm object is used in family modeling and massing in Revit.
- It is the base class for Sweep, Blend, Extrusion, and Revolution.

#### 10.2.2.1 Solid Form

SolidForms returns a read-only set of all solid forms representing solid parts in the main family body.

| |
|---|
| `Autodesk.Revit.Elements.GenericFormSet SolidForms {get;}` |

The main family body, Table-Dining Round w Chairs, is a round table composed of five basic parts (See the following picture). All five parts are the GenericForm type (Solid Extrusions in this case) whose IsSolid property is true. In this case, the property returns a set containing five Extrusion objects. If the main family body contains no solid forms, the property returns an empty set.

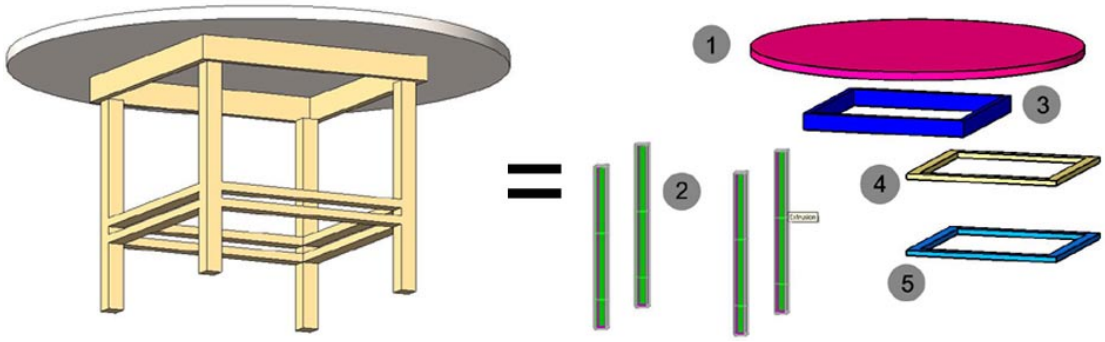



**Figure 81: Five parts (GenericForm objects) of the table**

#### 10.2.2.2 Void Form

VoidForms returns a read-only set of all void forms representing the void parts in the main family body.

```
Autodesk.Revit.Elements.GenericFormSet VoidForms {get;}
```

In the previous example, the main family body, Table-Dining Round w Chairs, with GenericForm type parts, contains no void forms. In this case, this property returns an empty set. If a family contains void forms, this property will return them.

For instance, in the following picture, the Family contains one solid form (Extrusion) as the main body, and one void form (also an Extrusion) cutting a cylindrical solid from the main body. The object in this Family has the SolidForms property and the VoidForms property whose sizes are both one.

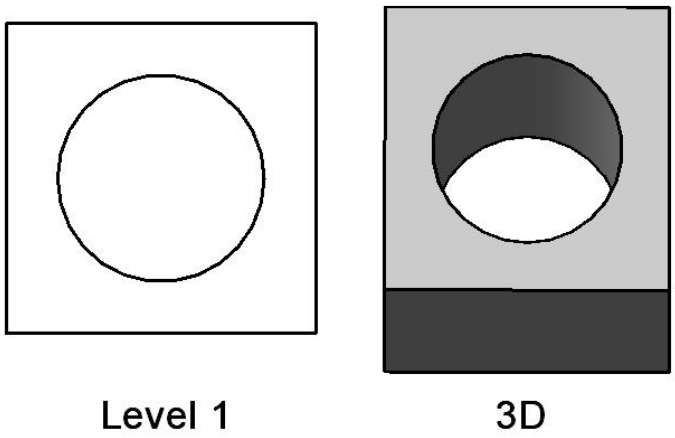



**Figure 82: An Element whose Family contains void forms**

### 10.2.3 Others

Others returns a read-only set of all other components in a particular family.

```
Autodesk.Revit.ElementSet Others {get;}
```

In the previous example, when retrieving the Others property from the Table-Dining Round w Chairs family, a set containing many kinds of Element objects is returned. Element objects include the following:

- Level
- LevelType
- Symbol
- ReferencePlane
- Dimension
- DimensionType
- TextNoteType
- View
- ViewPlan
- Sketch
- SketchPlan
- ModelLine, Group
- and more.

The objects are part of the Revit project and are used to retrieve information related to the family in the current project.

### 10.2.4 Load

The Document class contains the LoadFamily and LoadFamilySymbol methods.

- LoadFamily loads an entire family and all of its types or symbols into the project.
- LoadFamilySymbol loads only the specified family symbol from a family file into the project.

**Note**: Loading an entire family can take a long time and use a lot of memory. To avoid time and memory issues only load the symbols you need.

- The family file path is retrieved using the Options.Application object LibraryPaths property.
- The Options.Application object is retrieved using the Application object Options property.
- In LoadFamilySymbol, the input argument Name is the same string value returned by the FamilySymbol object Name property.

For more information, refer to the Code Samples in this chapter.

## 10.3 FamilyInstance

Examples of FamilyInstance objects in Revit are Beams, Braces, Columns, Furniture, Massing, and so on. The FamilyInstance object provides more detailed properties so that the family instance type and appearance in the project can be changed.

### 10.3.1 Location-Related Properties

Location-related properties show the physical and geometric characteristics of FamilyInstance objects, such as orientation, rotation and location.

#### 10.3.1.1 Orientation

The face orientation or hand orientation can be changed for some FamilyInstance objects. For example, a door can face the outside or the inside of a room or wall and it can be placed with the handle on the left side or the right side. The following table compares door, window, and desk family instances.

| Boolean Property | Door | Window (Fixed: 36”w x 72”h) | Desk |
|---|---|---|---|
| CanFlipFacing | True | True | False |
| CanFlipHand | True | False | False |

**Table 23: Compare Family Instances**

If CanFlipFacing or CanFlipHand is true, you can call the flipFacing or flipHand methods respectively. These methods can change the facing orientation or hand orientation respectively. Otherwise, the methods do nothing and return False.

When changing orientation, remember the following tips:

- In the Revit project, the user interface reminds you that an orientation can be flipped. An arrow with two directions appears in the middle of the element, shown in the following picture. There are two small arrows in the picture labeled 2.
  - One is for the facing orientation
  - One is for the hand orientation.
  - In the case of a window, you only see one glyph for the facing orientation.
- Some types of windows can change both hand orientation and facing orientation, such as a Casement 3x3 with Trim family.

There are four different facing orientation and hand orientation combinations for doors. See the following picture for the combinations and the corresponding Boolean values are in the following table.

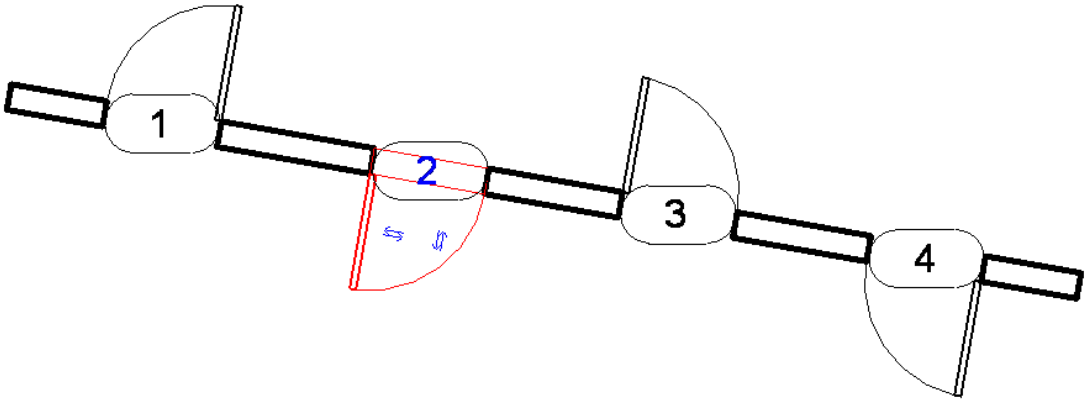



**Figure 83: Doors with different Facing and Hand Orientations**

| Boolean Property | Door 1 | Door 2 | Door 3 | Door 4 |
|---|---|---|---|---|
| FacingFlipped | False | True | False | True |
| HandFlipped | False | True | True | False |

**Table 24: Different Instances of the Same Type**

#### 10.3.1.2 Rotation - Mirrored

The Mirrored property indicates whether the FamilyInstance object has been mirrored.

| Boolean Property | Door 1 | Door 2 | Door 3 | Door 4 |
|---|---|---|---|---|
| Mirrored | False | False | True | True |

**Table 25: Door Mirrored Property**

In the previous door example, the Mirrored property for Door 1 and Door 2 is False. The Mirrored property for both Door 3 and Door 4 is True. It is because Door 1 and Door 2 are in the default state. If a user draws a door in the Revit project, the default result is either Door 1 or Door 2. To draw a door like Door 3 or Door 4, you must flip the Door 1 and Door 2 hand orientation respectively. The flip operation is like a mirror transformation. This is why the Door 3 and Door 4 Mirrored properties are True.

For more information about using the Mirror operation in Revit, refer to the Editing Elements chapter.

#### 10.3.1.3 Rotation – CanRotate and Rotate

The family instance Boolean CanRotate property is used to test whether the family instance can be rotated 180 degrees. This depends on the family to which the instance belongs. For example, in the following picture, the CanRotate properties for Window 1 (Casement 3x3 with Trim: 36”x72”) and Door 1 (Double-Glass 2: 72”x82”) are true, while Window 2 (Fixed: 36”w x 72”h) is false.

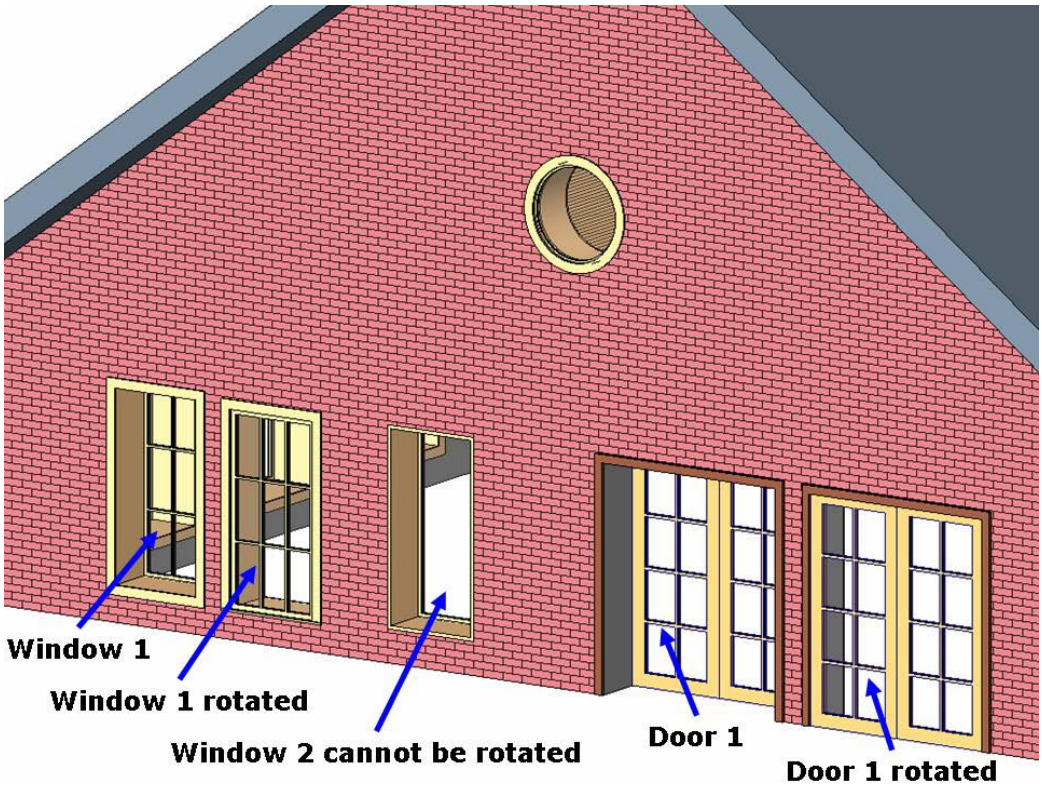



**Figure 84: Changes after rotate()**

If CanRotate is true, you can call the family instance Rotate method, which flips the family instance by 180 degrees. Otherwise, the method does nothing and returns False. The previous picture also shows the Window 1 and Door 1 states after executing the rotate method.

#### 10.3.1.4 Location

Location is used to find the physical location of an instance in a project. An instance can have a point location or a line location.

```
Autodesk.Revit.Location Location {get;}
```

The following characteristics apply to Location:

- A point location is a LocationPoint class object - A footing, a door, or a table has a point location
- A line location is a LocationCurve class object - A beam has a line location.
- They are both subclasses of the Location class.

For more information about Location, refer to the Editing Elements chapter.

### 10.3.2 Host and Component

Host and Component are both FamilyInstance properties. Component can be further divided into Subcomponent and Supercomponent.

#### 10.3.2.1 Host

A FamilyInstance object has a Host property that returns its hosting element.

```
Autodesk.Revit.Element Host {get;}
```

In the example of the round table, the property returns nothing because no host elements are created. An example of an instance that is hosted in another host element is a window or a door. In this case the Host property returns a wall in which the window or the door is located. See the following picture.

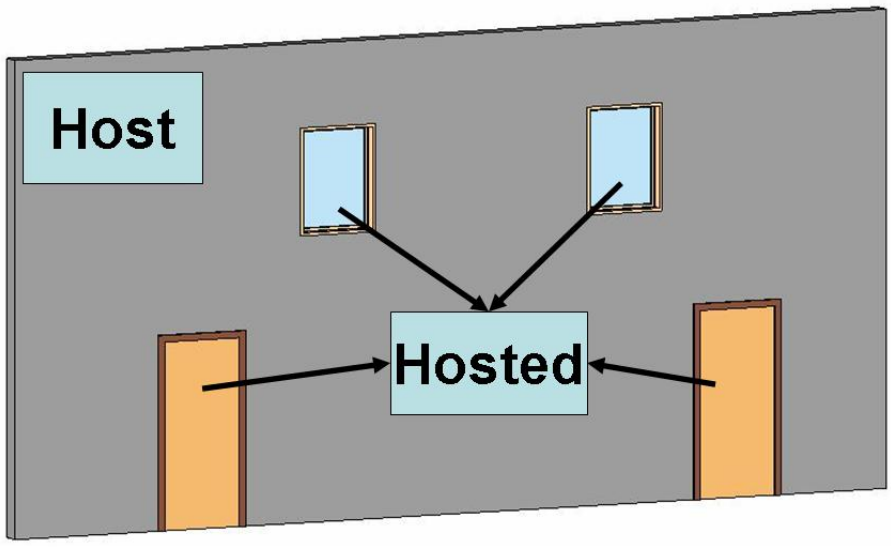



**Figure 85: Doors and windows hosted in a wall**

**Note**: Some FamilyInstance objects do not have host elements, such as Tables and other furniture. However, other subjects, such as doors and windows, must have host elements to be in the Revit project.

#### 10.3.2.2 Subcomponent and Supercomponent

The SubComponent property returns the family instance auxiliary components.

```
Autodesk.Revit.ElementSet SubComponents {get;}
```

In the Table-Dining Round w Chairs.rfa example, the chairs around the table are returned by the SubComponent property. The difference between SubComponents and Family.Components is as follows:

- SubComponents returns the current symbol components
- Family.Components returns the default symbol

The SuperComponent property returns the family instance parent component. In the Table-Dining Round w Chairs example, the family instance supercomponent for one chair is the table.

```
Autodesk.Revit.Element SuperComponent {get;}
```

### 10.3.3 Other Properties

The properties in this section are specific to Revit Architecture and Revit Structure. They are covered thoroughly in their respective chapters.

#### 10.3.3.1 Room Information

FamilyInstance properties include Room, FromRoom, and ToRoom. For more information about Room, refer to the Revit Architecture chapter.

#### 10.3.3.2 Revit Structure Related Analytical Model

AnalyticalModel retrieves the family instance structural analytical model. Different analytical models are returned based on the instance type.

```
Autodesk.Revit.Structural.AnalyticalModel AnalyticalModel {get;}
```

For example, if the instance is a beam, a frame analytical model is returned. If the instance is a footing, a location analytical model is returned. For more information about AnalyticalModel refer to the Revit Structure chapter.

### 10.3.4 Creation

The Document.NewFamilyInstance method is used to create a new FamilyInstance object in the Revit project.

Some FamilyInstance objects require more than one location to be created. In these cases, it is more appropriate to use the more detailed creation method provided by this object. If the instance is not created, an exception is thrown. The type/symbol used must be loaded into the project before the method is called.

Families and family symbols are loaded using the Document.LoadFamily or Document.LoadFamilySymbol methods. Some families, such as Beams, have more than one endpoint and are inserted in the same way as a single point instance. Once the linear family instances are inserted, their endpoints can be changed using the Element.Location property. For more information, refer to the Code Samples section in this chapter.

## 10.4 Code Samples

Review the following code samples for more information about working with Component Elements. Please note that in the NewFamilyInstance method, a StructuralType argument is required to specify the type of the family instance to be created. Here are some examples:

| Type of Family Instance | Value of StructuralType |
|---|---|
| Doors, tables, etc. | NonStructural |
| Beams | Beam |
| Braces | Brace |
| Columns | Column |
| Footings | Footing |

**Table 26: The value of StructuralType argument in the NewFamilyInstance method**

### 10.4.1 Create Tables

The following function demonstrates how to load a family of Tables into a Revit project and create instances from all symbols in this family.

The LoadFamily method returns false if the specified family was previously loaded. Therefore, in the following case, do not load the family, Table-Dining Round w Chairs.rfa, before this function is called. In this example, the tables are created at Level 1 by default.

**Code Region 1-ComponentElements**

```
void CreateTables(Autodesk.Revit.Application revit)
{
   // ...
   // Use the family file: …\Furniture\Table-Dining Round w
   // Chairs.rfa";
   Autodesk.Revit.Elements.Family family = null;
   if (!document.LoadFamily(fileName, ref family))
   {
      return;
   }
   FamilySymbolSetIterator symbolItor = family.Symbols.ForwardIterator();
   int x = 0, y = 0;
   while (symbolItor.MoveNext())
   {
      FamilySymbol symbol = symbolItor.Current as FamilySymbol;
      XYZ location = new XYZ(x, y, 10);
      // Do not use the overloaded NewFamilyInstance method that contains
      // the Level argument, otherwise Revit cannot show the instances
      // correctly in 3D View, for the table is not level-based component.
      FamilyInstance instance = document.Create.NewFamilyInstance(ref location, symbol,
       StructuralType.NonStuctural);
      x += 10;
   }
}
```

The result of loading the Tables family:

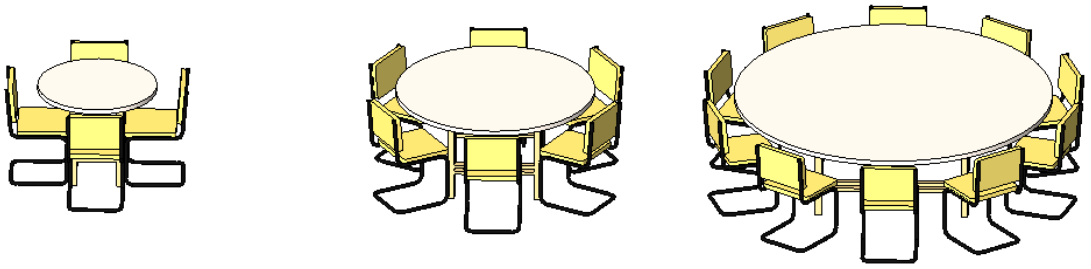


**Figure 86: Load family and create tables in the Revit project**

### 10.4.2 Create a Beam

In this sample, a family symbol is loaded instead of a family, because loading the family, M_W-Wide Flange.rfa, takes a long time. However, loading a symbol such as W1100X499 is very fast. In the end, the beam location curve is changed to make it normal.

**Code Region 2-ComponentElements**

```
void CreateBeam(Autodesk.Revit.Application revit)
{
   // ...
   // Using the family file: …\Structural\Framing\Steel\M_W-Wide
   // Flange.rfa, and using the Symbol name: W1100X499.
   FamilySymbol gotSymbol = null;
   if (document.LoadFamilySymbol(fileName, name, ref gotSymbol))
   {
      XYZ location = new XYZ(0, 0, 0);
      FamilyInstance instance = document.Create.NewFamilyInstance(ref location, gotSymbol,
       level, StructuralType.Beam);
      LocationCurve curveLocation = instance.Location as LocationCurve;
      if (null != curveLocation)
      {
         XYZ endPoint = new XYZ(10, 0, 0);
         curveLocation.Curve = Line.get_Bound(ref location, ref endPoint);
      }
   }
}
```

The picture on the right is the result if you use the code as written. If the gray lines in the previous code are commented out, the beam curve start point and end point are identical (by default, length equals 0). The picture on the left is the result if you comment out the code indicated.

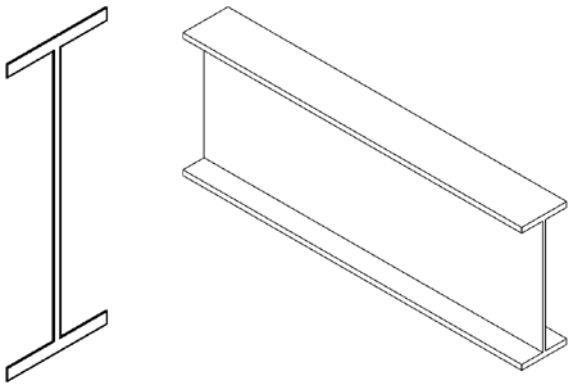


**Figure 87: Result of not changing (left) and changing (right) the Location property**

### 10.4.3 Create Doors

Create a long wall about 180'0'' in length and select it before running this sample. The host object must support inserting instances; otherwise the NewFamilyInstance method will fail. If a host element is not provided for an instance that must be created in a host, or the instance cannot be inserted into the specified host element, the method NewFamilyInstance does nothing.

**Code Region 3-ComponentElements**

```
void CreateDoorsInWall(Autodesk.Revit.Application revit)
{
   // ...
   // Use the family file: …\Doors\Single-Decorative 2.rfa
   Autodesk.Revit.Elements.Family family = null;
   if (!document.LoadFamily(fileName, ref family))
   {
      return;
   }
   FamilySymbolSetIterator symbolItor = family.Symbols.ForwardIterator();
   double x = 0, y = 0, z = 0;
   while (symbolItor.MoveNext())
   {
      FamilySymbol symbol = symbolItor.Current as FamilySymbol;
      XYZ location = new XYZ(x, y, z);
      FamilyInstance instance = document.Create.NewFamilyInstance(ref location, symbol, wall,
       level, StructuralType.NonStuctural);
      x += 10;
      y += 10;
      z += 1.5;
   }
}
```

The result of the previous code in Revit is shown in the following picture. Notice that if the specified location is not at the specified level, the NewFamilyInstance method uses the location elevation instead of the level elevation.

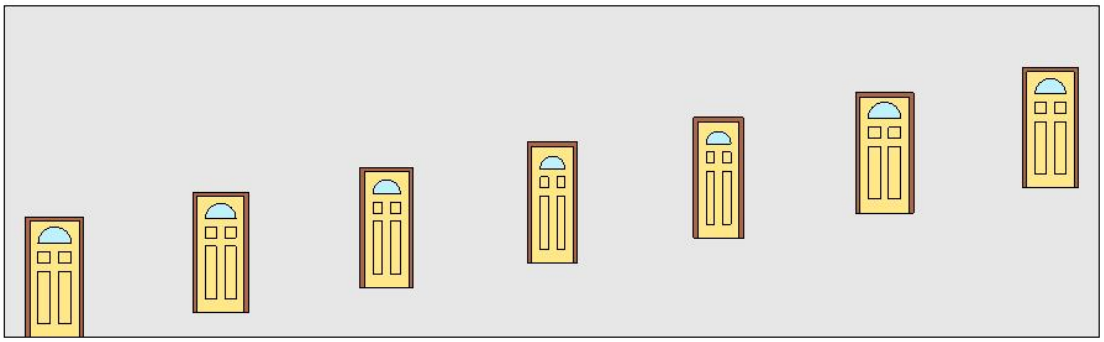


**Figure 88: Insert doors into a wall**

### 10.4.4 Create FamilyInstances Using Reference Directions

Use reference direction to insert an item in a specific direction.

**Code Region 4-ComponentElements**

```
void CreatenIstancesReferDirection(Autodesk.Revit.Application revit)
{
   // ...
   // Use the family file: …\Furniture\Bed-Box.rfa.
   Autodesk.Revit.Elements.Family family = null;
   if (!document.LoadFamily(fileName, ref family))
   {
      return;
   }
   FamilySymbolSetIterator symbolItor = family.Symbols.ForwardIterator();
   int x = 0, y = 0;
   int i = 0;
   while (symbolItor.MoveNext())
   {
      FamilySymbol symbol = symbolItor.Current as FamilySymbol;
      XYZ location = new XYZ(x, y, 0);
      XYZ direction = new XYZ();
      switch (i % 3)
      {
         case 0:
            direction = new XYZ(1, 1, 0);
            break;
         case 1:
            direction = new XYZ(0, 1, 1);
            break;
         case 2:
            direction = new XYZ(1, 0, 1);
            break;
      }
      FamilyInstance instance = document.Create.NewFamilyInstance(ref location, symbol, ref
       direction, floor, StructuralType. NonStuctural);
```

```
      x += 10;
      i++;
   }
}
```

The result of the previous code appears in the following picture:

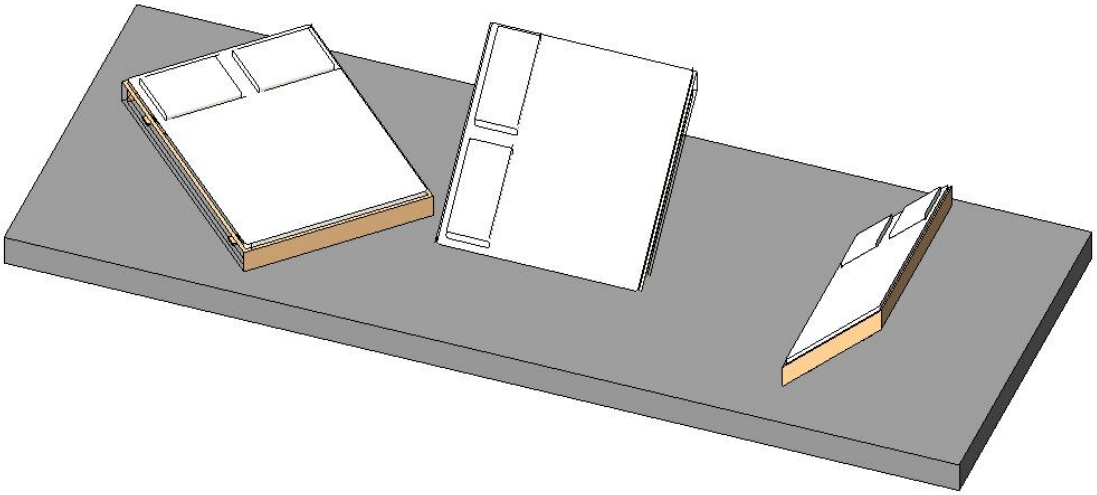


**Figure 89: Create family instances using different reference directions**

# 11 Datum and Information Elements

This chapter introduces Datum Elements and Information Elements in Revit.

- Datum Elements include levels, grids, ModelCurve and so on.
- Information Elements include phases, design options, gbXMLParamElem and so on.

For more information about Revit Element classifications, refer to the Elements Essentials chapter

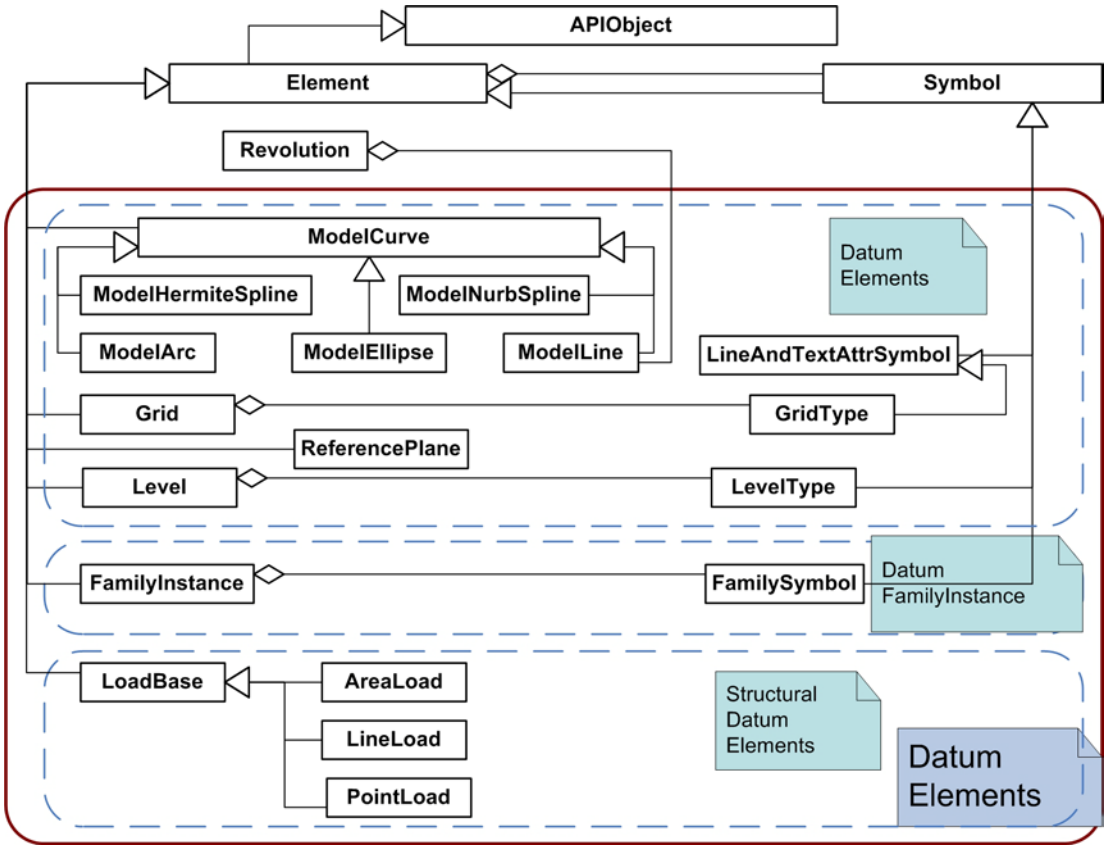



**Figure 90: Datum Elements diagram**

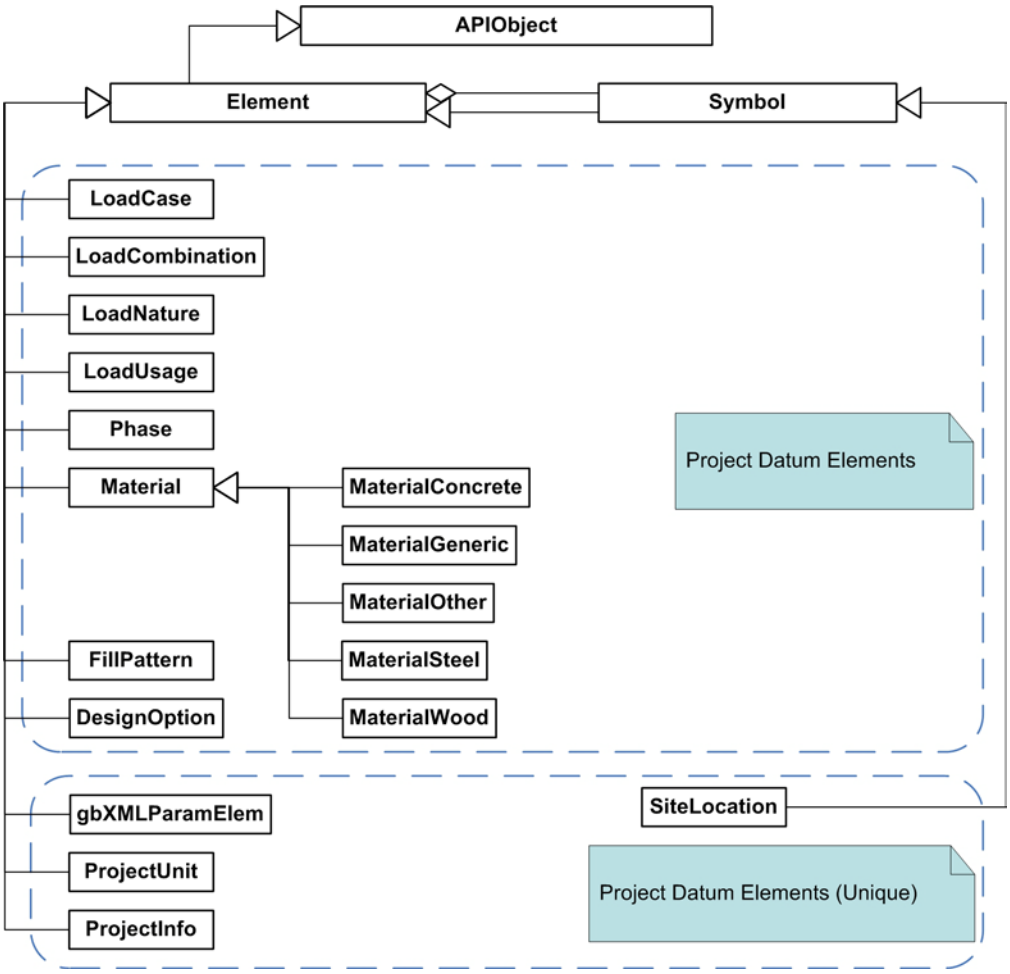



**Figure 91: Information Elements diagram**

**Note:** Many elements in the previous diagram are not introduced in this chapter. If you need more information, refer to the related chapter:

- For LoadBase, LoadCase, LoadCombination, LoadNature and LoadUsage, refer to the Revit Structure chapter
- For ModelCurve, refer to the Sketching chapter
- For Material and FillPattern, refer to the Material chapter
- For gbXMLParamElem, refer to the Revit Architecture chapter

# 11.1 Level

A level is a finite horizontal plane that acts as a reference for level-hosted elements, such as roofs, floors, and ceilings. In the Revit API, the Level class is derived from the Element class. The inherited Name property is used to retrieve the user-visible level name beside the level bubble in the Revit UI. To retrieve all levels in a project, use the ElementIterator iterator to search for Level objects.

## 11.1.1 Elevation

The Level class has the following properties:

- The Elevation property (LEVEL_ELEV) is used to retrieve or change the elevation above or below ground level.

- The ProjectElevation property is used to retrieve the elevation relative to the project origin regardless of the Elevation Base parameter value.
- Elevation Base is a Level type parameter.
  - Its BuiltInParameter is LEVEL_RELATIVE_BASE_TYPE.
  - Its StorageType is Integer
  - 0 corresponds to Project and 1 corresponds to Shared.

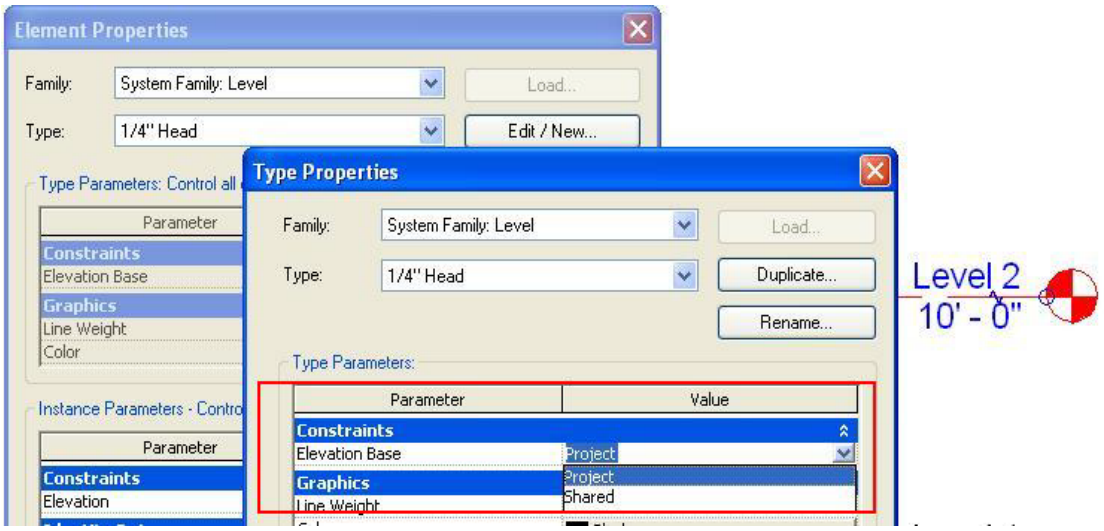



**Figure 92: Level Type Elevation Base property**

When you use the Tools > Shared Coordinates > Acquire Coordinates command to get the coordinates of a linked project in Revit, the coordinates in the host project are updated. The following changes are made when the coordinates are updated:

- The shared coordinate's origin in the linked project becomes the new origin for the updated coordinates in the host project.
- The Elevations of all levels in the host project are updated based on the new coordinates if the value of the Elevation Base parameter is set to Shared.
- The Project Elevations remain the same because they are based on the internal coordinates in the host project.

**Note:** If you want to get the new elevation levels after acquiring the coordinates from a linked RVT project, set the Elevation Base parameter value to Shared. Otherwise the Elevation and Project Elevation values are equal.

The following code sample illustrates how to retrieve all levels in a project using ElementIterator.

**Code Region 1 - Datum and Information Elements**

```
public IExternalCommand.Result Execute(ExternalCommandData commandData, ref string message,
  ElementSet elements)
{
   //Before invoke this sample, please link a .RVT file into the current project
   //and use the Tools > Shared Coordinate > Acquire Coordinate command to get
   //the coordinate of the linked project.
   Document doc = commandData.Application.ActiveDocument;

   ElementIterator elemItor = doc.Elements;
   elemItor.Reset();
```

```
   while (elemItor.MoveNext())
   {
      Level aLevel = elemItor.Current as Level;
      if (null == aLevel)
      {
         continue;
      }

      //Get the elevation and project elevation of the level.
      double projectElevation = aLevel.ProjectElevation;
      double elevation = aLevel.Elevation;

      //Get the level type
      LevelType levelType = aLevel.LevelType;
      Parameter levelBaseParam =
       levelType.get_Parameter(BuiltInParameter.LEVEL_RELATIVE_BASE_TYPE);
      int levelBaseValue = levelBaseParam.AsInteger();
      string levelbaseString = "Impossible go here.";

      //try to change the value of Elevation Base prameter to see the
      //different of Elevation value.
      if (1 == levelBaseValue)
      {
         levelbaseString = "Shared";
      }
      else if (0 == levelBaseValue)
      {
         levelbaseString = "Project";
         Debug.Assert(projectElevation == elevation);

         levelBaseParam.Set(1);
         double realElevation = aLevel.Elevation;
         levelBaseParam.Set(0);
      }
   }
   return IExternalCommand.Result.Succeeded;
}
```

### 11.1.2 Create a Level

Using the Level command, you can define a vertical height or story within a building and you can create a level for each existing story or other building references. Levels must be added in a section or elevation view. Additionally, you can create a new level using the Revit API.

The following code sample illustrates how to create a new level.

**Code Region 2 - Datum and Information Elements**

```
Level CreateLevel(Document doc)
```

```
{
   // The elevation to apply to the new level
   double elevation = 20.0;
   // Begin to create a level
   Level level = doc.Create.NewLevel(elevation);
   if (null == level)
   {
      throw new Exception("Create a new level failed.");
   }
   //check whether the level name is unique and change the level name
  string newName = "New level";
  bool nameIsUnique = true;
  ElementIterator iter = doc.Elements;
  iter.Reset();
  while (iter.MoveNext())
  {
      Level temp = iter.Current as Level;
      if (null != temp && temp.Name == newName)
      {
         nameIsUnique = false;
        break;
      }
   }
  if (nameIsUnique)
   {
      level.Name = newName;
  }
   return level;
}
```

**Note**: After creating a new level, Revit does not create the associated plan view for this level. If necessary, you can create it yourself. For more information about how to create a plan view, refer to the Views chapter.

## 11.2 Grid

Grids are represented by the Grid class which is derived from the Element class. It contains all grid properties and methods. The inherited Name property is used to retrieve the content of the grid line's bubble.

### 11.2.1 Curve

The Grid class Curve property gets the object that represents the grid line geometry.

- If the IsCurved property returns true, the Curve property will be an Arc class object.
- If the IsCurved property returns false, the Curve property will be a Line class object.

For more information, refer to the Geometry chapter.

The following code is a simple example using the Grid class. The result appears in a message box after invoking the command.

**Code Region 3 - Datum and Information Elements**

```
public void GetInfo_Grid(Grid grid)
{
   string message = "Grid : ";
   // Show IsCurved property
   message += "\nIf grid is Arc : " + grid.IsCurved;
   // Show Curve information
   Autodesk.Revit.Geometry.Curve curve = grid.Curve;
   if (grid.IsCurved)
   {
      // If the curve is an arc, give center and radius information
      Autodesk.Revit.Geometry.Arc arc = curve as Autodesk.Revit.Geometry.Arc;
      message += "\nArc's radius: " + arc.Radius;
      message += "\nArc's center: " + XYZToString(arc.Center);
   }
   else
   {
      // If the curve is a line, give length information
      Autodesk.Revit.Geometry.Line line = curve as Autodesk.Revit.Geometry.Line;
      message += "\nLine's Length" + line.Length;
   }
   // Get curve start point
   message += "\nStart point: " + XYZToString(curve.get_EndPoint(0));
   // Get curve end point
   message += "; End point: " + XYZToString(curve.get_EndPoint(1));
   MessageBox.Show(message, "Revit", MessageBoxButtons.OK);
}
// Output the point's three coordinates
string XYZToString(Autodesk.Revit.Geometry.XYZ point)
{
   return "(" + point.X + ", " + point.Y + ", " + point.Z + ")";
}
```

### 11.2.2 Create a Grid

Two overloaded methods are available to create a new grid in the Revit API. Using the following method with different parameters, you can create a curved or straight grid:

```
public Grid NewGrid( Arc arc );
public Grid NewGrid( Line line );
```

**Note:** The arc or the line used to create a grid must be in a horizontal plane.

The following code sample illustrates how to create a new grid with a line or an arc.

**Code Region 4 - Datum and Information Elements**

```
void CreateGrid(Autodesk.Revit.Application app)
{
   // Create the geometry line which the grid locates
   XYZ start = new XYZ(0, 0, 0);
   XYZ end = new XYZ(30, 30, 0);
   Line geomLine = Line.get_Bound(ref start, ref end);

   // Create the geometry arc which the grid locates
   XYZ end0 = new XYZ(0, 0, 0);
   XYZ end1 = new XYZ(10, 40, 0);
   XYZ pointOnCurve = new XYZ(5, 7, 0);
   Arc geomArc = app.Create.NewArc(ref end0, ref end1, ref pointOnCurve);

   // Create a grid using the geometry line
   Grid lineGrid = app.ActiveDocument.Create.NewGrid(geomLine);
   if (null == geomArc)
   {
      throw new Exception("Create a new straight grid failed.");
   }
   // Create a grid using the geometry arc
   Grid arcGrid = app.ActiveDocument.Create.NewGrid(geomArc);
   if (null == arcGrid)
   {
      throw new Exception("Create a new curved grid failed.");
   }
}
```

**Note**: In Revit, the grids are named automatically in a numerical or alphabetical sequence when they are created.

## 11.3 Phase

Some architectural projects, such as renovations, proceed in phases. Phases have the following characteristics:

- Phases represent distinct time periods in a project lifecycle.
- The lifetime of an element within a building is controlled by phases.
- Each element has a construction phase but only the elements with a finite lifetime have a destruction phase.

All phases in a project can be retrieved from the Document object. A Phase object contains three pieces of useful information: Name, ID and UniqueId. The remaining properties always return null or an empty collection.

You can define phases and phase filters in your project, and then choose the phase to show and the filter to apply to the view. If desired, you can make multiple copies of a view and then apply different phases and phase filters to the different copies.

**Figure 93: Phases in a project**

Each new modeling component added to a project has a Phase Created and a Phase Demolished property.The Phase Created property has the following characteristics:

- It identifies the phase in which the component was added.
- The default value is the same as the current view Phase value.
- Change the Phase Created parameter by selecting a new value from the drop-down list.

The Phase Demolished property has the following characteristics:

- It identifies in which phase the component is demolished.
- The default value is none.
- Demolishing a component with the demolition tool updates the property to the current Phase value in the view where you demolished the element.
- You can demolish a component by setting the Phase Demolished property to a different value.
- If you delete a phase using the Revit API, all modeling components in the current phase still exist. The Phase Created parameter value for these components is changed to the next item in the drop-down list in the Properties dialog box.

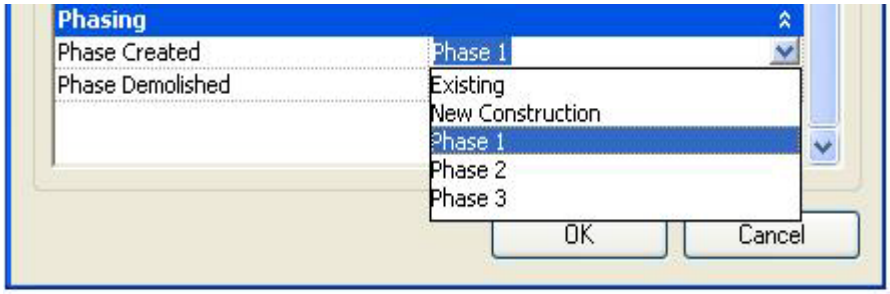



**Figure 94: Phase-created component parameter value**

The following code sample displays all supported phases in the current document. For practice, add some phases in Revit and then invoke the code sample. The phase names are displayed in a message box.

**Code Region 5 - Datum and Information Elements**

```
void Getinfo_Phase(Document doc)
{
   // Get the phase array which contains all the phases.
   PhaseArray phases = doc.Phases;
```

```
   // Format the prompt string which identifies all supported phases in the current document.
   String prompt = null;
   if (0 != phases.Size)
   {
      prompt = "All the phases in current document list as follow:";
      foreach (Phase ii in phases)
      {
         prompt += "\n\t" + ii.Name;
      }
   }
   // Give the user the information.
   MessageBox.Show(prompt, "Revit", MessageBoxButtons.OK);
}
```

## 11.4 Design Options

Design options provide a way to explore alternative designs in a project. Design options provide the flexibility to adapt to changes in project scope or to develop alternative designs for review. You can begin work with the main project model and then develop variations along the way to present to a client. Most elements can be added into a design option. Elements that cannot be added into a design option are considered part of the main model and have no design alternatives.

The main use for Design options is as a property of the Element class. See the following example.

**Code Region 6 - Datum and Information Elements**

```
void Getinfo_DesignOption(Document document)
{
   // Get the Element in the Active Document
   Autodesk.Revit.ElementIterator elementiter = document.Elements;

   elementiter.Reset();
   while(elementiter.MoveNext())
   {
      Autodesk.Revit.Element element = elementiter.Current as Autodesk.Revit.Element;

      //Use the DesignOption property of Element
      if(element.DesignOption != null)
      MessageBox.Show(element.DesignOption.Name.ToString());
   }
}
```

The following rules apply to Design Options

- The value of the DesignOption property is null if the element is in the Main Model. Otherwise, the name you created in the Revit UI is returned.
- Only one active DesignOption Element can exist in an ActiveDocument.
  - The primary option is considered the default active DesignOption. For example, a design option set is named Wall and there are two design options in this set named “brick wall”

and “glass wall”. If “brick wall” is the primary option, only this option and elements that belong to it are retrieved by the Element Iterator. “Glass wall” is inactive.

– To activate an inactive design option, select Tools > Design Options > Pick to Edit to pick a secondary option, or make the secondary option primary using Tools > Design Options > Design Options….

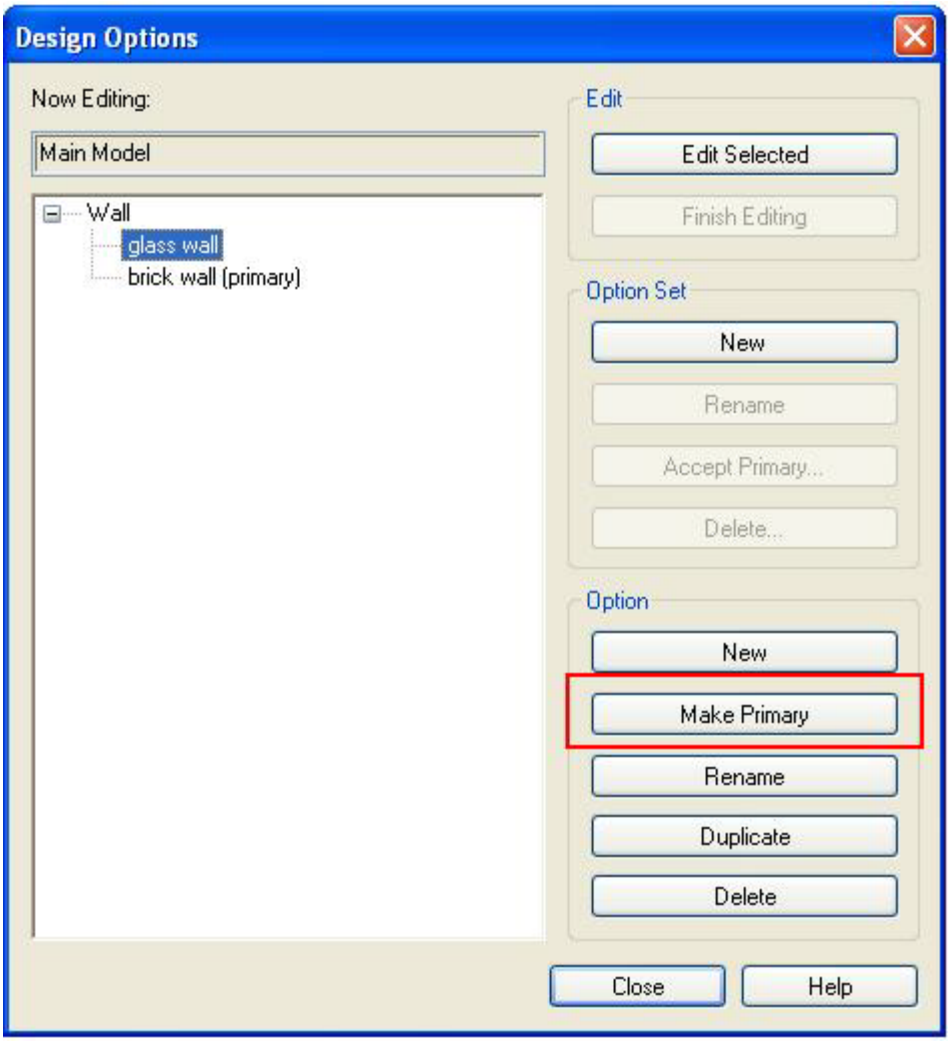



**Figure 95 : Design Options**

# 12 Annotation Elements

This chapter introduces Revit Annotation Elements, including the following:

- Dimension
- DetailCurve
- IndependentTag
- TextNote
- AnnotationSymbol
- and more

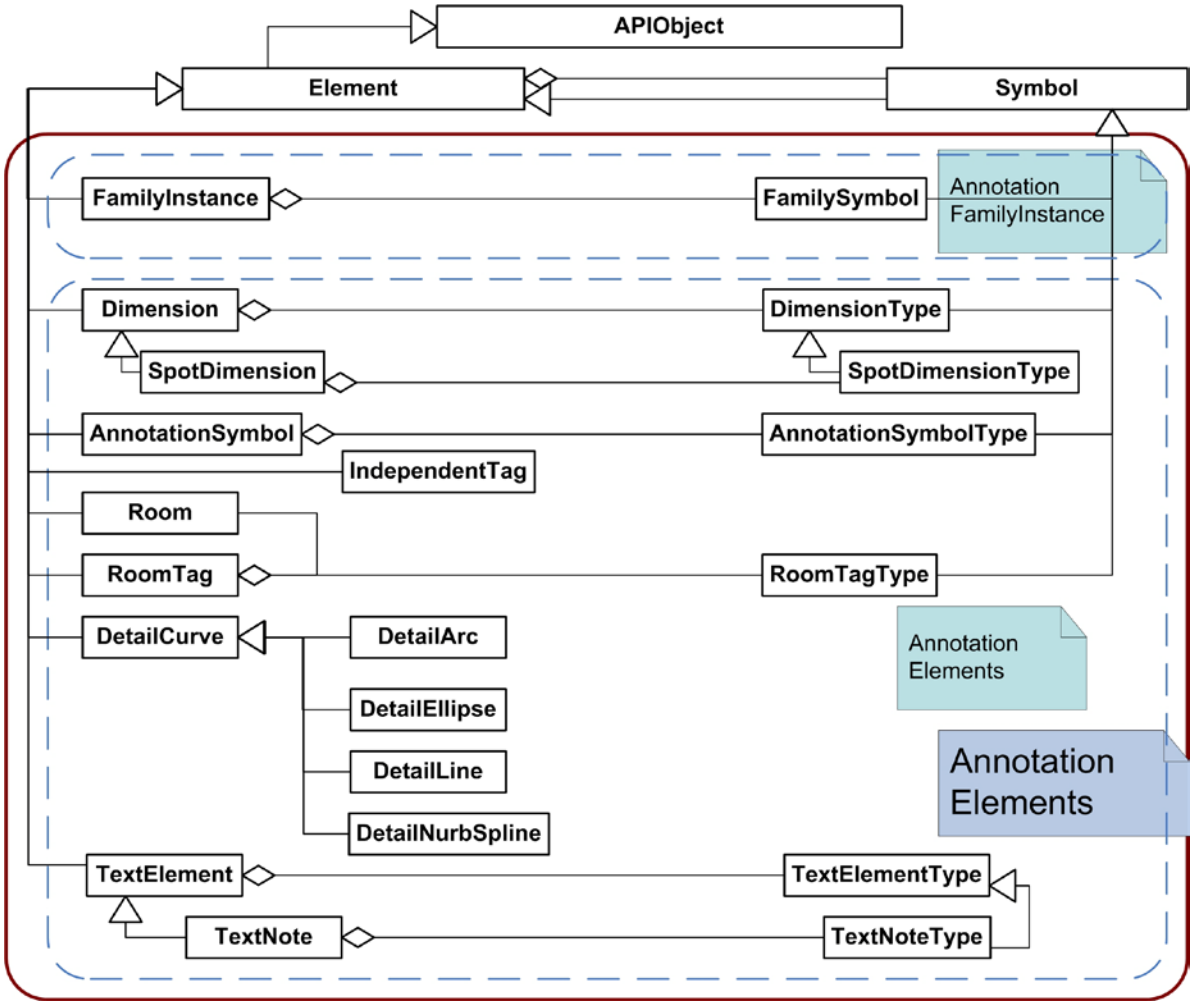



**Figure 96: Annotation Elements diagram**

- Dimensions are view-specific elements that display sizes and distances in a project.
- Detail curves are created for detailed drawings. They are visible only in the view in which they are drawn. Often they are drawn over the model view.
- Tags are an annotation used to identify elements in a drawing. Properties associated with a tag can appear in schedules.
- AnnotationSymbol has multiple leader options when loaded into a project.

For more information about Revit Element classification, refer to the Elements Essentials chapter.

## 12.1 Dimension and Constraint

The Dimension class represents permanent dimensions and dimension related constraint elements.

- Only non-transient elements are accessible using the Revit API and temporary dimensions are transient.
- Spot elevation and spot coordinate are represented by the SpotDimension class.

The following code sample illustrates how to distinguish permanent dimensions from constraint elements.

**Code Region 1 - Annotation Elements**

```
//Dimension dimension = GetDimension();
if ((int)BuiltInCategory.OST_Dimensions == dimension.Category.Id.Value)
{
   // it's a permanent dimension
}
else if ((int)BuiltInCategory.OST_Constraints == dimension.Category.Id.Value)
{
   // it's a constraint elements
}
```

### 12.1.1 Permanent Dimensions

There are four kinds of permanent dimensions:

- Linear dimension
- Radial dimension
- Angular dimension
- Arc length dimension

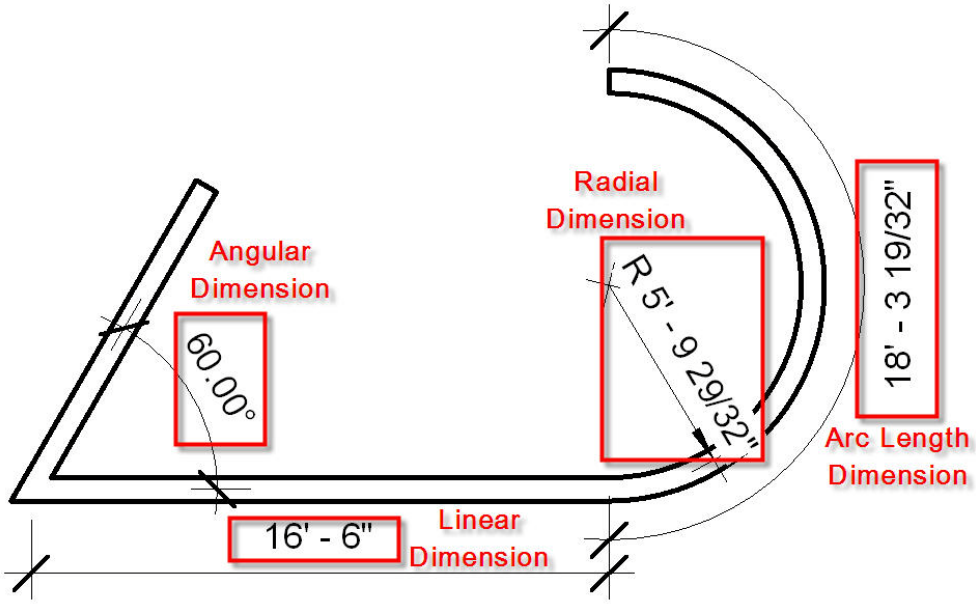



**Figure 97: Permanent dimensions**

The BuiltInCategory for all permanent dimensions is OST_Dimensions. Currently, there is not an easy way to distinguish the four dimensions using the API.

**Note:** Temporary Dimensions are not accessible by the API because they are not persistent objects.

Except for radial dimension, every dimension has one dimension line. Dimension lines are available from the Dimension.Curve property which is always unbound. In other words, the dimension line does not have a start-point or end-point. Based on the previous picture:

- A Line object is returned for a linear dimension.
- An arc object is returned for a radial dimension or angular dimension.
- A radial dimension returns null.

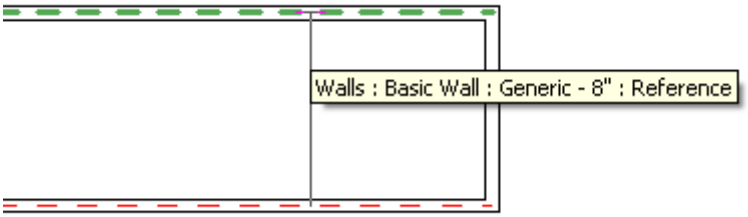



**Figure 98: Dimension references**

A dimension is created by selecting geometric references as the previous picture shows. Geometric references are represented as a Reference class in the API. The following dimension references are available from the References property. For more information about Reference, please see the Geometry.Reference section in the Geometry chapter.

- Radial dimension - One Reference object for the curve is returned
- Angular and arc length dimensions - Two Reference objects are returned.
- Linear dimensions - Two or more Reference objects are returned. In the following picture, the linear dimension has five Reference objects.

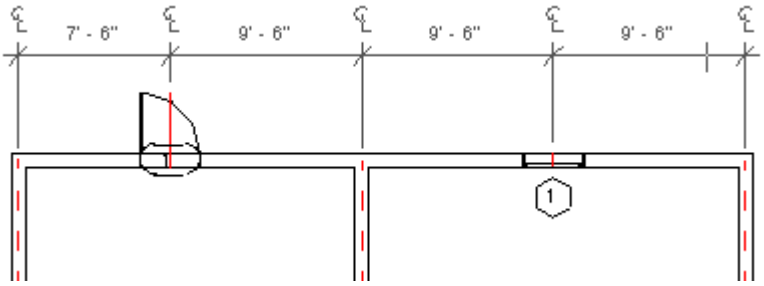



**Figure 99: Linear dimension references**

Dimension, like other Annotation Elements, is view-specific. It displays only in the view where it is added. The View property returns the specific view.

### 12.1.2 Constraint Elements

Dimension objects with Category Constraints (BuitInCategory.OST_Constraints) represent two kinds of dimension-related constraints:

- Linear and radial dimension constraints
- Equality constraints

In the following picture, two kinds of locked constraints correspond to linear and radial dimension. In the application, they appear as padlocks with green dashed lines. (The green dashed line is

available from the Dimension.Curve property.) Both linear and radial dimension constraints return two Reference objects from the Dimension.References property.

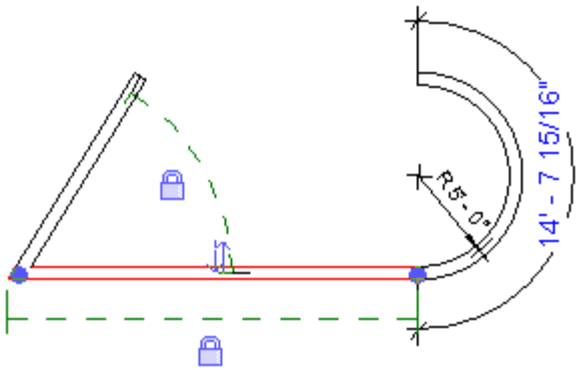



**Figure 100: Linear and Radial dimension constraints**

Constraint elements are not view-specific and can display in different views. Therefore, the View property always returns null. In the following picture, the constraint elements in the previous picture are also visible in the 3D view.

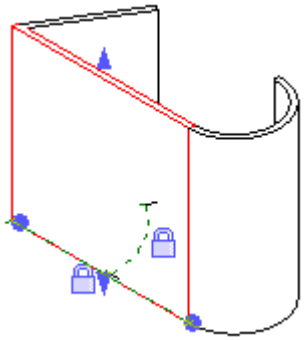


**Figure 101: Linear and Radial dimension constraints in 3D view**

Although equality constraints are based on dimensions, they are also represented by the Dimension class. There is no direct way to distinguish linear dimension constraints from equality constraints in the API using a category or DimensionType. Equality constraints return three or more References while linear dimension constraints return two or more References.

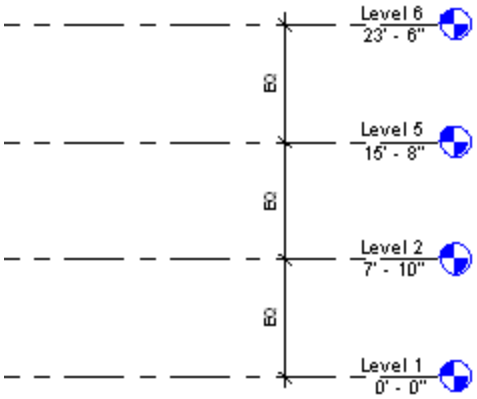



**Figure 102: Equality constraints**

To show the constraint element curve in the previous picture in Revit, complete the following steps:

1. Lock a permanent linear or radial dimension in the Revit application.
2. Delete the dimension. A warning dialog box appears.
3. Click OK to close the warning dialog box.
4. Select the element to which the dimension refers.

**Note**: Not all constraint elements are represented by the Dimension class but all belong to a Constraints (OST_Constraints) category such as alignment constraint.

### 12.1.3 Spot Dimensions

Spot coordinates and spot elevations are represented by the SpotDimension class and are distinguished by category. Like the permanent dimension, spot dimensions are view-specific. The type and category for each spot dimension are listed in the following table:

| Type | Category |
|---|---|
| Spot Coordinates | OST_SpotCoordinates |
| Spot Elevations | OST_SpotElevations |

**Table 27: Spot dimension Type and Category**

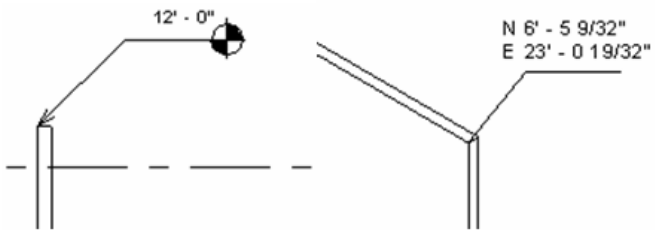



**Figure 103: SpotCoordinates and SpotElevations**

The SpotDimension Location can be downcast to LocationPoint so that the point coordinate that the spot dimension points to is available from the LocationPoint.Point property.

- SpotDimensions have no dimension curve so their Curve property always returns null.
- The SpotDimension References property returns one Reference representing the point or the edge referenced by the spot dimension.
- To control the text and tag display style, modify the SpotDimension and SpotDimensionType Parameters.

### 12.1.4 Comparison

The following table compares different kinds of dimensions and constraints in the API:

| Dimension or Constraint | | API Class | BuiltInCategory | Curve | Reference | View | Location |
|---|---|---|---|---|---|---|---|
| Permanent Dimension | linear dimension | Dimension | OST_Dimensions | A Line | >=2 | Specific view | null |
| | radial dimension | | | Null | 1 | | |
| | angular dimension | | | An Arc | 2 | | |
| | arc length dimension | | | An Arc | 2 | | |

| Dimension or Constraint | | API Class | BuiltInCategory | Curve | Reference | View | Location |
|---|---|---|---|---|---|---|---|
| Dimension Constraint | linear dimension constraint | | OST_Constraints | A Line | 2 | | |
| | angular dimension | | | An Arc | 2 | | |
| Equality Constraint | | | | A Line | >=3 | | |
| Spot Dimension | Spot Coordinates | SpotDimension | OST_SpotCoordinates | Null | 1 | Specific view | LocationPoint |
| | Spot Elevations | | OST_SpotElevations | | | | |

**Table 28: Dimension Category Comparison**

### 12.1.5 Create and Delete

The create dimension method is available in the Creation.Document class. Currently, this method is applied to create a linear dimension only.

```
public Dimension NewDimension (View view, Line line, ReferenceArray references)
public Dimension NewDimension (View view, Line line, ReferenceArray references,
   DimensionType dimensionType)
```

Using the NewDimension method input parameters, you can define the visible View, dimension line, and References (two or more). However, there is no easy way to distinguish a linear dimension DimensionType from other types. Currently, the overloaded NewDimension method with the DimensionType parameter is rarely used.

The following code illustrates how to use the NewDimension method to duplicate a dimension.

**Code Region 2 - Annotation Elements**

```
//Dimension dimension = GetDimension();
Line line = dimension.Curve as Line;
if (null != line)
{
   View view = dimension.View;
   ReferenceArray references = dimension.References;
   Autodesk.Revit.Creation.Document docCreator =
   commandData.Application.ActiveDocument.Create;
   Dimension newDimension = docCreator.NewDimension(view, line, references);
}
```

Though only linear dimensions are created, you can delete all dimensions and constraints represented by Dimension and SpotDimension using the Document.Delete method.

## 12.2 Detail Curve

Detail curve is an important Detail component usually used in the detail or drafting view. Detail curves are accessible in the DetailCurve class and its derived classes.

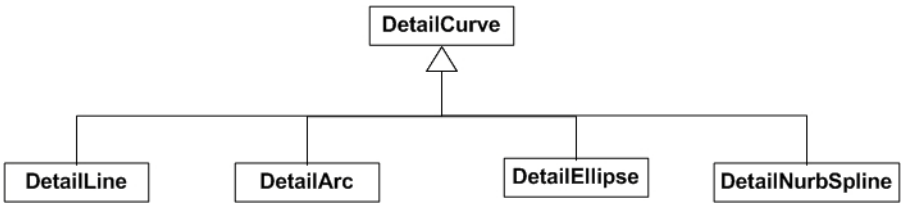



**Figure 104: DetailCurve diagram**

DetailCurve is view-specific as are other annotation elements. However, currently there is no DetailCurve.View property. When creating a detail curve, you must compare the detail curve to the model curve view.

```
public DetailCurve NewDetailCurve (View, Curve, SketchPlane)
public ModelCurve NewModelCurve (Curve, SketchPlane)
```

Generally only 2D views such as level view and elevation view are acceptable, otherwise an exception is thrown.

Except for view-related features, DetailCurve is very similar to ModelCurve. For more information about ModelCurve properties and usage, see the ModelCurve section in the Sketching chapter.

## 12.3 Tags

A tag is an annotation used to identify drawing elements. The API exposes the IndependentTag and RoomTag classes to cover most tags used in the Revit application. For more details about RoomTag, see the Room section in the Revit Architecture chapter.

**Note**: The IndependentTag class represents the tag element in Revit and other specific tags such as keynote, beam system tag, electronic circuit symbol (Revit MEP), and so on. In Revit internal code, the specific tags have corresponding classes derived from IndependentTag. As a result, specific features are not exposed by the API and cannot be created using the NewTag method. They can be distinguished by the following categories:

| Tag Name | BuiltInCategory |
|---|---|
| Keynote Tag | OST_KeynoteTags |
| Beam System Tag | OST_BeamSystemTags |
| Electronic Circuit Tag | OST_ElectricalCircuitTags |
| Span Direction Tag | OST_SpanDirectionSymbol |
| Path Reinforcement Span Tag | OST_PathReinSpanSymbol |
| Rebar System Span Tag | OST_IOSRebarSystemSpanSymbolCtrl |

**Table 29: Tag Name and Category**

In this section, the main focus is on the tag type represented in the following picture.

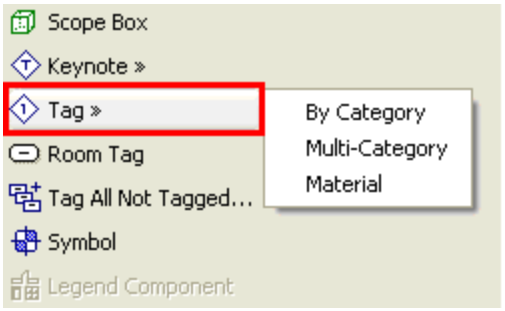



**Figure 105: IndependentTag**

Every category in the family library has a pre-made tag. Some tags are automatically loaded with the default Revit application template, while others are loaded manually. The IndependentTag objects return different categories based on the host element if it is created using the By Category option. For example, the Wall and Floor IndependentTag are respectively OST_WallTags and OST_FloorTags.

If the tag is created using the Multi-Category or Material style, their categories are respectively OST_MultiCategoryTags and OST_MaterialTags.

Similar to DetailCurve, NewTag only works in the 2D view, otherwise an exception is thrown. The following code is an example of IndependentTag creation. Run it when the level view is the active view.

**Code Region 3 - Annotation Elements**

```
//Wall wall = GetWall();
Autodesk.Revit.Elements.View view = commandData.Application.ActiveDocument.ActiveView;
if (ViewType.ThreeD == view.ViewType)
   return;

TagMode tagMode = TagMode.TM_ADDBY_CATEGORY;
TagOrientation tagorn = TagOrientation.TAG_HORIZONTAL;

// Add the tag to the middle of the wall
LocationCurve wallLoc = wall.Location as LocationCurve;
XYZ wallStart = wallLoc.Curve.get_EndPoint(0);
XYZ wallEnd = wallLoc.Curve.get_EndPoint(1);
XYZ wallMid = wallStart.Add(ref wallEnd).Divide(2);

Autodesk.Revit.Creation.Document docCreator = commandData.Application.ActiveDocument.Create;
IndependentTag newTag = docCreator.NewTag(view, wall, true, tagMode, tagorn, wallMid);
if (null == newTag)
{
  throw new Exception("Create IndependentTag Failed.");
}

newTag.TagText = "Hello";
// set leader mode free
// otherwise leader end point move with elbow point
```

```
newTag.LeaderMode = LeaderEndCondition.LEC_FREE;
XYZ elbowPnt = wallMid + new XYZ(5.0, 5.0, 0.0);
newTag.LeaderElbow = elbowPnt;
XYZ headerPnt = wallMid + new XYZ(10.0, 10.0, 0.0);
newTag.TagHeadPosition = headerPnt;
```

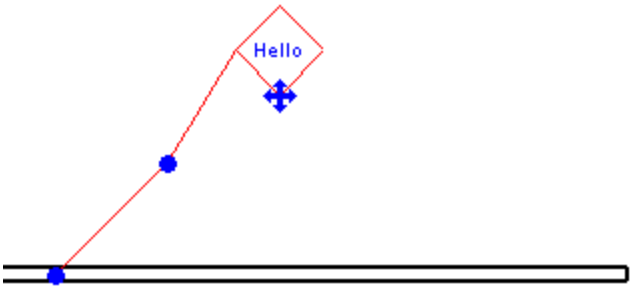



**Figure 106: Create IndependentTag using sample code**

## 12.4 Text

In the API, the TextNote class represents Text. Its general features are as follows:

| Function | API Method or Property |
|---|---|
| Add text to the application | Creation.Document.NewTextNote |
| Get and set string from the text component | Does not exist in this version. |
| Get and set text component position | TextNote.Coord Property |
| Get and set text component width | TextNote.Width Property |
| Get all text component leaders | TextNote.Leaders Property |
| Add a leader to the text component | TextNote.AddLeader Method |
| Remove all leaders from the text component | TextNote.RemoveLeaders Method |

**Table 30: General Features of TextNote**

Revit supports two kinds of Leaders: straight leaders and arc leaders. Control the TextNote leader type using the TextNoteLeaderType enumerated type:

| Function | Member Name |
|---|---|
| —Add a right arc leader | TNLT_ARC_R |
| —Add a left arc leader | TNLT_ARC_L |
| —Add a right leader. | TNLT_STRAIGHT_R |
| —Add a left leader. | TNLT_STRAIGHT_L |

**Table 31: Leader Types**

**Note**: Straight leaders and arc leaders cannot be added to a Text type at the same time.

## 12.5 Annotation Symbol

An annotation symbol is a symbol applied to a family to uniquely identify that family in a project.

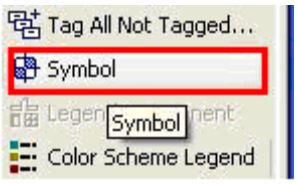



**Figure 107: Add annotation symbol**

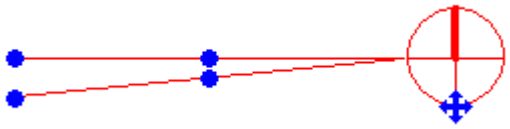


**Figure 108: Annotation Symbol with two leaders**

### 12.5.1 Create and Delete

The annotation symbol creation method is available in the Creation.Document class.

```
public Dimension NewAnnotationSymbol(ref XYZ location, FamilySymbol symbol, Level level)
```

The annotation symbol can be deleted using the Document.Delete method.

### 12.5.2 Add and Remove Leader

Add and remove leaders using the addLeader and removeLeader methods.

**Code Region 4 - Annotation Elements**

```
//AnnotationSymbol symbol = GetSymbol();
int leaderSize = symbol.Leaders.Size;
symbol.addLeader();
leaderSize = symbol.Leaders.Size;
symbol.removeLeader();
leaderSize = symbol.Leaders.Size;
```

# 13 Sketching

To create elements or edit their profiles in Revit, you must first sketch them in a sketch editor. Examples of elements that require sketches include:

- Roofs
- Floors
- Stairs
- Railings.

Sketches are also required to define other types of geometry, such as:

- Extrusions
- Openings
- Regions

In the Revit API, sketch functions are represented by 2D and 3D sketch classes such as the following:

- 2D sketch class include:
  - SketchPlane
  - Sketch
  - ModelCurve
  - and more
- 3D sketch class include:
  - GenericForm
  - Path3D
  - and more

In addition to Sketch Elements, ModelCurve is also included in this chapter. For more details about Element Classification, see the Elements Classification section in the Elements Essentials chapter.

## 13.1 2D Sketch Class

The Sketch class represents enclosed curves in a plane used to create a 3D model. The key features are represented by the SketchPlane and CurveLoop properties.

When editing a Revit file, you cannot retrieve a Sketch object by iterating Document.Elements enumeration because all Sketch objects are transient Elements. When accessing the Family's 3D modeling information, Sketch objects are important to forming the geometry. For more details, refer to the 3D Sketch section in this chapter.

The following picture shows the key classes to sketch in 2D.

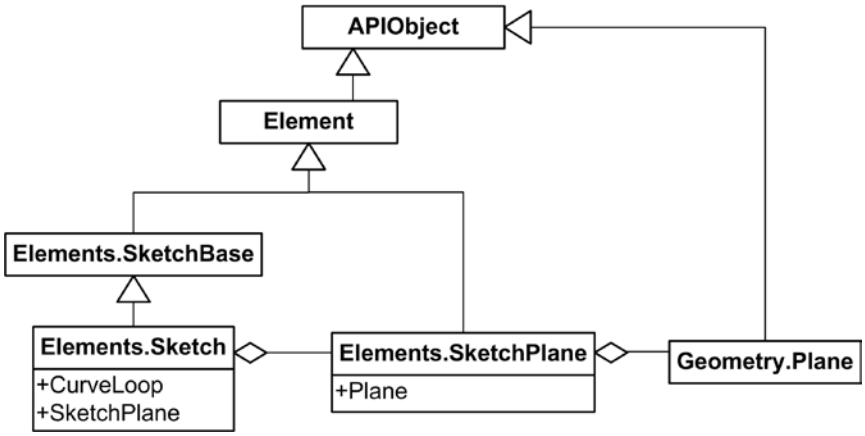



**Figure 109: 2D Sketch diagram**

SketchPlane is the basis for all 2D sketch classes such as ModelCurve and Sketch. SketchPlane is also the basis for 2D Annotation Elements such as DetailCurve. Both ModelCurve and DetailCurve have the SketchPlane property and need a SketchPlane in the corresponding creation method.

SketchPlane is always invisible in the Revit UI. To test its visibility, complete the following steps:

1. Get the ID of a SketchPlane by iterating Document.Elements.
2. Run Revit and from the menubar select Tools > Element Ids > Select by Id. Enter the ID of a SketchPlane to try to display it by its ID. A message dialog box appears with the message, No good view could be found.

Every ModelCurve must lie in one SketchPlane. In other words, wherever you draw a ModelCurve either in the UI or by using the API, a SketchPlane must exist. Therefore, at least one SketchPlane exists in a 2D view where a ModelCurve is drawn.

The 2D view contains the CeilingPlan, FloorPlan, and Elevation ViewTypes. By default, a SketchPlane is automatically created for all of these views. The 2D view-related SketchPlane Name returns the view name such as Level 1 or North.

**Figure 110: Pick a Plane to identify a new Work Plane**

When you specify a new work plane, you can select Pick a plane as illustrated in the previous picture. After you pick a plane, select a plane on a particular element such as a wall as the following picture shows. In this case, the SketchPlane.Name property returns a string related to

that element. For example, in the following picture, the SketchPlane.Name property returns ‘Generic - 8’ the same as the Wall.Name property.

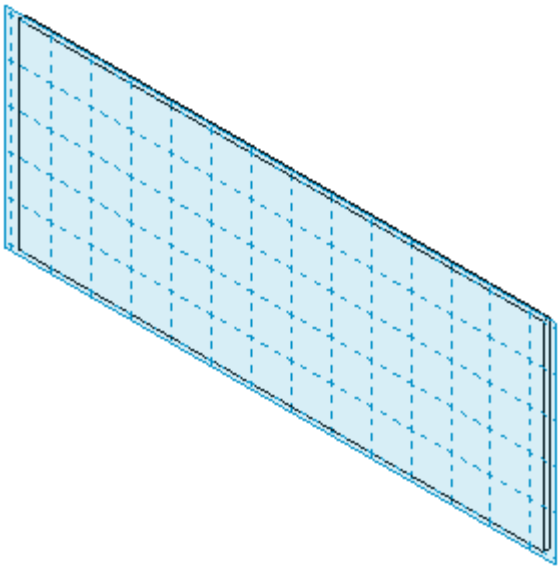


**Figure 111: Pick a Plane on a wall as Work Plane**

**Note:** Sketchplane is different from work plane because work plane is visible and can be selected. It does not have a specific class in the current API but is represented by the Element class. A work plane must be defined based on a specific SketchPlane. Both the work plane and SketchPlane Category property return null. Although SketchPlane is always invisible, there is always a SketchPlane that corresponds to a work plane. A work plane is used to express a SketchPlane in text and pictures.

The following information applies to SketchPlane members:

- ID, UniqueId, Name, and Plane properties return a value;
- Parameters property is empty
- Location property returns a Location object
- Others return null.

Plane contains the SketchPlane geometric information. SketchPlane sets up a plane coordinate system with Plane as the following picture illustrates:

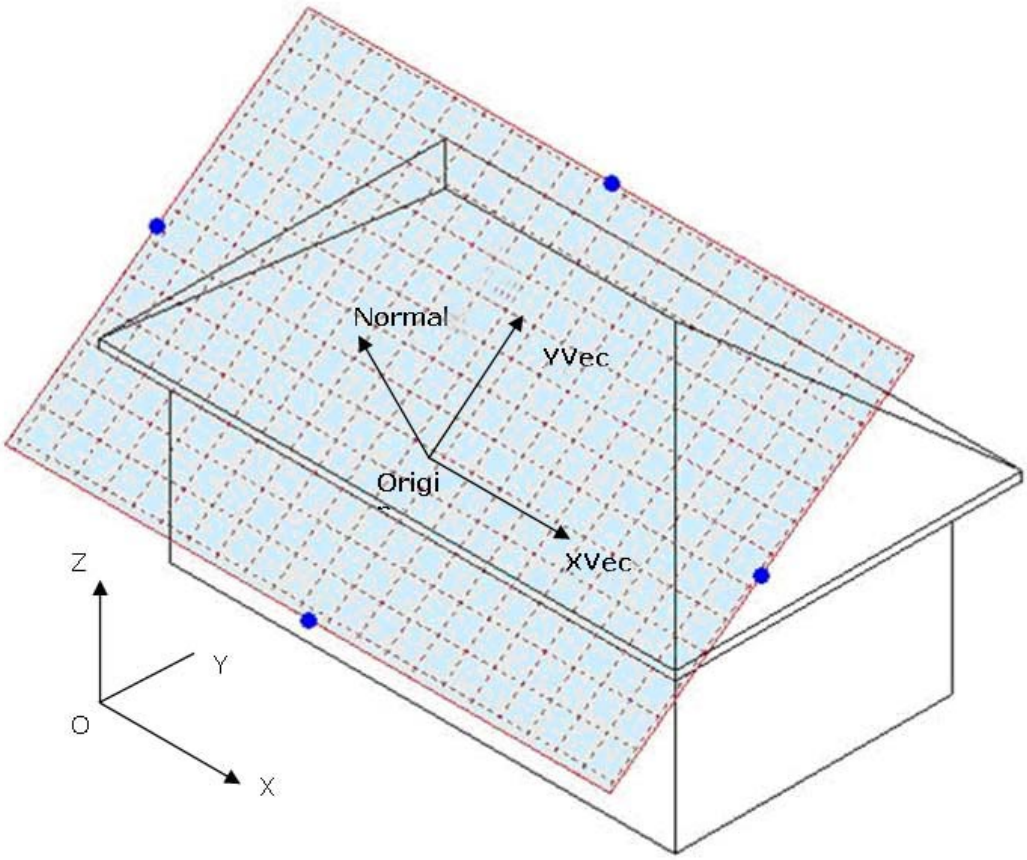



**Figure 112: SketchPlane and Plane coordinate system**

The following code sample illustrates how to create a new SketchPlane:

**Code Region 1 - Sketching**

```
//…Other operation

Autodesk.Revit.Application app = commandData.Application;

// Create a new sketch plane
XYZ newNormal = new XYZ(1, 1, 0);  // the normal vector
XYZ newOrigin = new XYZ(0, 0, 0);  // the origin point

// Create geometry plane
Plane geometryPlane = app.Create.NewPlane(ref newNormal,ref newOrigin);

// Create sketch plane
SketchPlane sketchPlane = app.ActiveDocument.Create.NewSketchPlane(geometryPlane);

//…Other operation
```

## 13.2 3D Sketch

3D Sketch is used to edit a family or create a 3D object. In the Revit UI, you can complete a 3D Sketch with a Solid Form or a Void Form as follows:

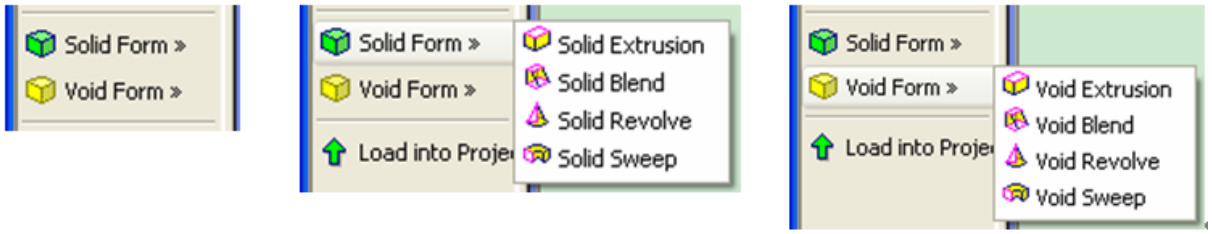



**Figure 113: Solid Form and Void Form**

In the Revit API, you can complete the 3D Sketch using the following classes. However, the 3D model can only be accessed and not created through the Revit API.

- Extrusion
- Revolution
- Blend
- Sweep

In other words, there are four operations through which a 2D model turns into a 3D model. For more details about sketching in 2D, refer to the 2D Sketch section in this chapter. The classes derive from the Generic Form which is retrieved using the Family.SolidForms property and the Family.VoidForms property. For more details about Generic Form, refer to the Solid and Void Forms section in the Component Elements chapter.

### 13.2.1 Extrusion

Revit Architecture uses extrusions to define 3D geometry for families. You create an extrusion by defining a 2D sketch on a plane; Revit Architecture then extrudes the sketch between a start and an end point.

Query the Extrusion Form object for a generic form to use in family modeling and massing. The Extrusion class has the following properties:

| Property | Description |
|---|---|
| ExtrusionStart | Returns the Extrusion Start point. It is a Double type. |
| ExtrusionEnd | Returns the Extrusion End point. It is a Double type. |
| Sketch | Returns the Extrusion Sketch. It contains a sketch plane and some curves. |

**Table 32: Extrusion Properties**

The ExtrusionStart and ExtrusionEnd properties value is consistent with the parameters in the Revit UI. The following pictures illustrate the corresponding parameters and the Extrusion result.

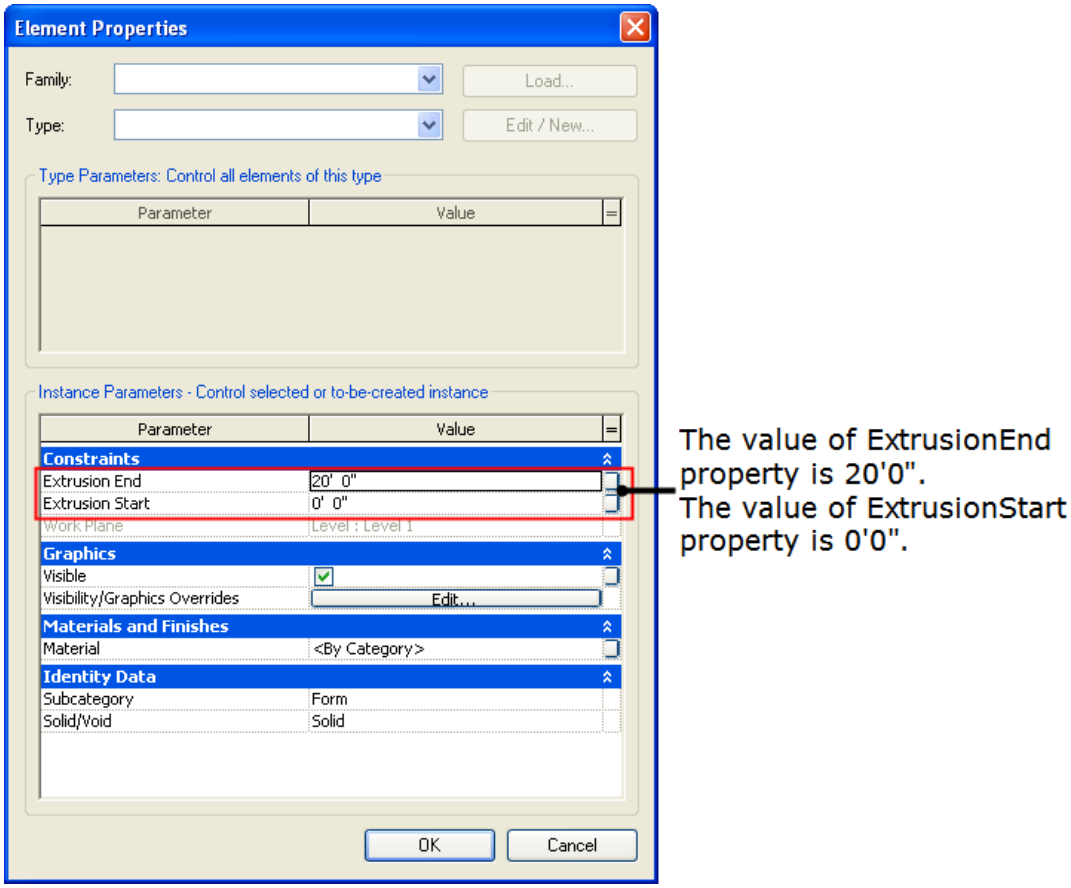



**Figure 114: Corresponding parameter**

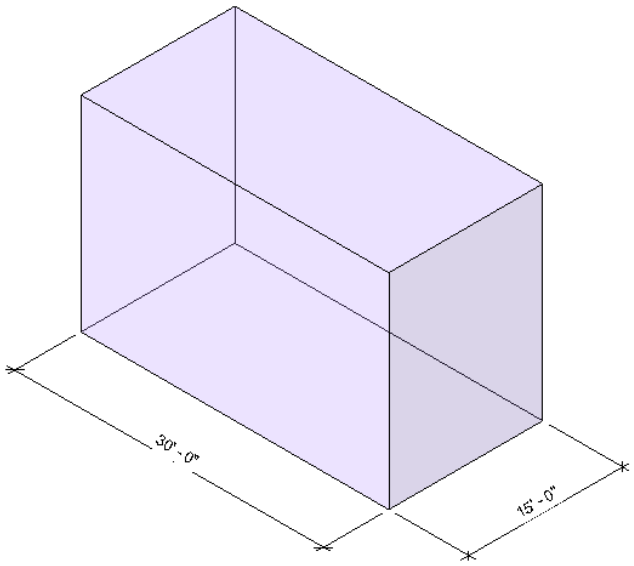



**Figure 115: Extrusion result**

### 13.2.2 Revolution

The Revolve command creates geometry that revolves around an axis. You can use the revolve command to create door knobs or other knobs on furniture, a dome roof, or columns.

Query the Revolution Form object for a generic form to use in family modeling and massing. The Revolution class has the following properties:

| Property | Description |
|---|---|
| Axis | Returns the Axis. It is a ModelLine object. |
| EndAngle | Returns the End Angle. It is a Double type. |
| Sketch | Returns the Extrusion Sketch. It contains a SketchPlane and some curves. |

**Table 33: Revolution Properties**

EndAngle is consistent with the same parameter in the Revit UI. The following pictures illustrate the Revolution corresponding parameter, the sketch, and the result.

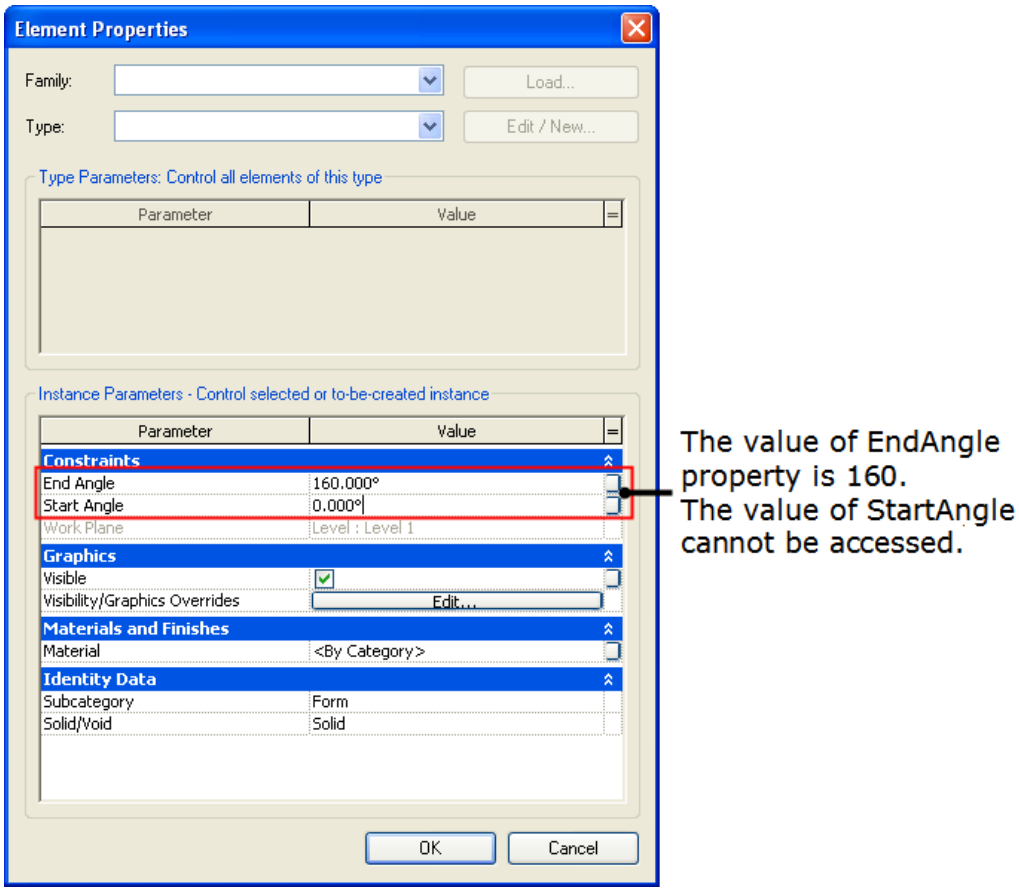



**Figure 116: Corresponding parameter**

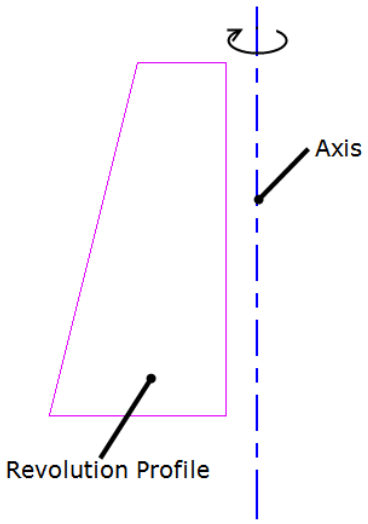



**Figure 117: Revolution sketch**

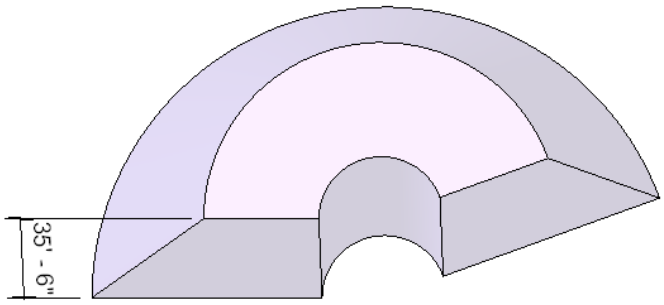



**Figure 118: Revolution result**

**Note:**

- The Start Angle is not accessible using the Revit API.
- If the End Angle is positive, the Rotation direction is clockwise. If it is negative, the Rotation direction is counterclockwise

### 13.2.3 Blend

The Blend command blends two profiles together. For example, if you sketch a large rectangle and a smaller rectangle on top of it, Revit Architecture blends the two shapes together.

Query the Blend Form object for a generic form to use in family modeling and massing. The Blend class has the following properties:

| Property | Description |
|---|---|
| BottomSketch | Returns the Bottom Sketch. It is a Sketch object. |
| TopSketch | Returns the Top Sketch Blend. It is a Sketch object. |
| FirstEnd | Returns the First End. It is a Double type. |
| SecondEnd | Returns the Second End. It is a Double type. |

**Table 34: Blend Properties**

The FirstEnd and SecondEnd property values are consistent with the same parameters in the Revit UI. The following pictures illustrate the Blend corresponding parameters, the sketches, and the result.

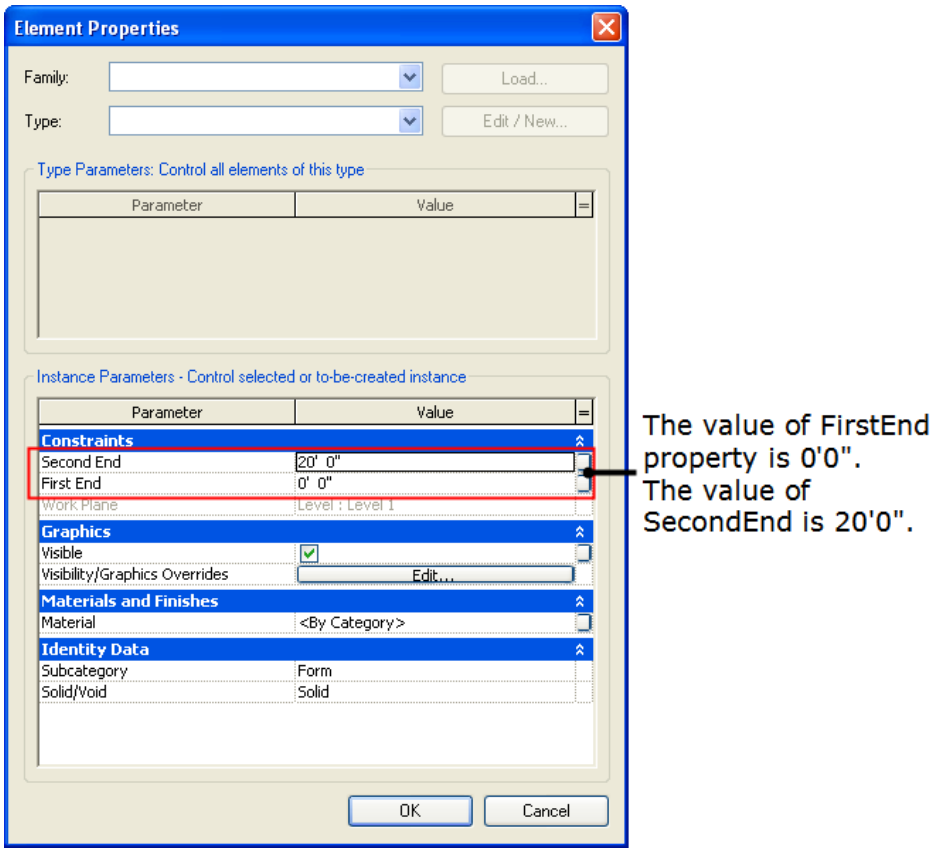



**Figure 119: Corresponding parameters**

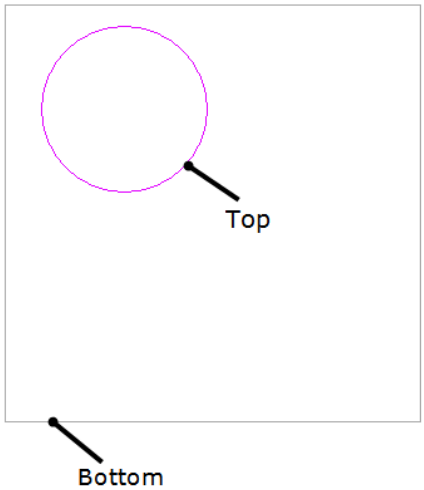



**Figure 120: Blend top sketch and bottom sketch**

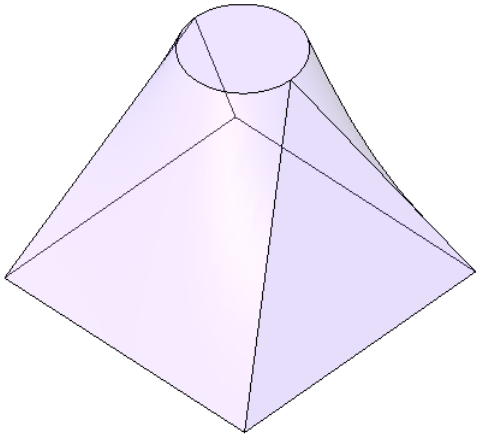


**Figure 121: Blend result**

### 13.2.4 Sweep

The Sweep command sweeps one profile along a created 2D path or selected 3D path. The path may be an open or closed loop, but must pierce the profile plane.

Query the Sweep Form object for a generic form for use in family modeling and massing. The Sweep class has the following properties:

| Property | Description |
|---|---|
| Path3d | Returns the 3D Path Sketch. It is a Path3D object. |
| PathSketch | Returns the Plan Path Sketch. It is a Sketch object. |
| ProfileSketch | Returns the profile Sketch. It is a Sketch object. |
| EnableTrajSegmentation | Returns the Trajectory Segmentation state. It is a Boolean. |
| MaxSegmentAngle | Returns the Maximum Segment Angle. It is a Double type. |

**Table 35: Sweep Properties**

Click the corresponding command to enter the path sketch editor. See the following example:

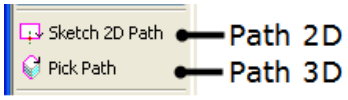



**Figure 122: 2D path and 3D path**

Creating a 2D Path is similar to other forms. The 3D Path is fetched by picking the created 3D curves.

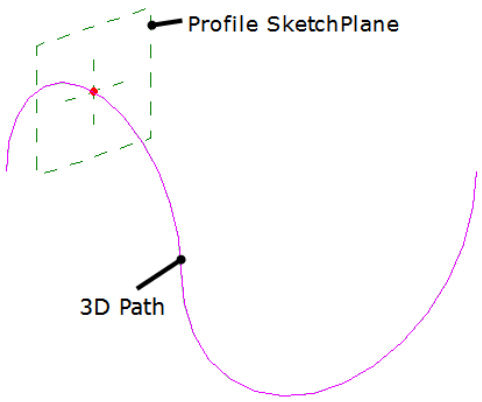



**Figure 123: Pick the Sweep 3D path**

**Note:** The following information applies to Sweep:

- The Path3d property is available only when you use Pick Path to get the 3D path.
- PathSketch is available whether the path is 3D or 2D.

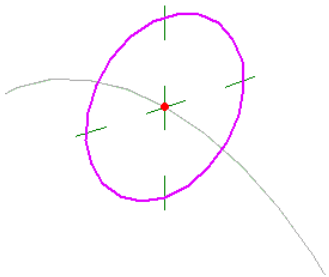


**Figure 124: Sweep profile sketch**

**Note:** The ProfileSketch is perpendicular to the path.

Segmented sweeps are useful for creating mechanical duct work elbows. Create a segmented sweep by setting two sweep parameters and sketching a path with arcs.

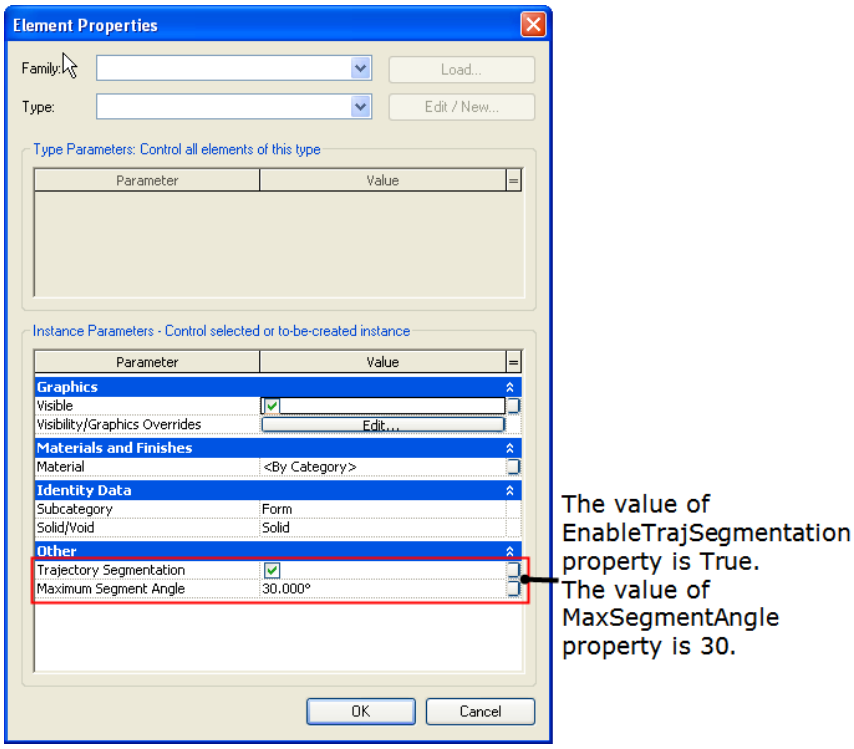



**Figure 125: Corresponding segment setting**

**Note:** The following information applies to segmented Sweeps:

- The parameters affect only arcs in the path.
- The minimum number of segments for a sweep is two.
- Change a segmented sweep to a non-segmented sweep by clearing the Trajectory Segmentation check box. The EnableTrajSegmentation property returns false.
- If the EnableTrajSegmentation property is false, the value of MaxSegmentAngle is the default 360°.

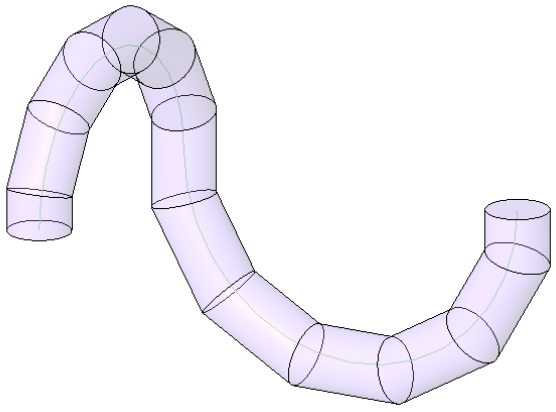


**Figure 126: Sweep result**

## 13.3 ModelCurve

ModelCurve represents model lines in the project. It exists in 3D space and is visible in all views.

The following picture shows the ModelCurve inheritance hierarchy.

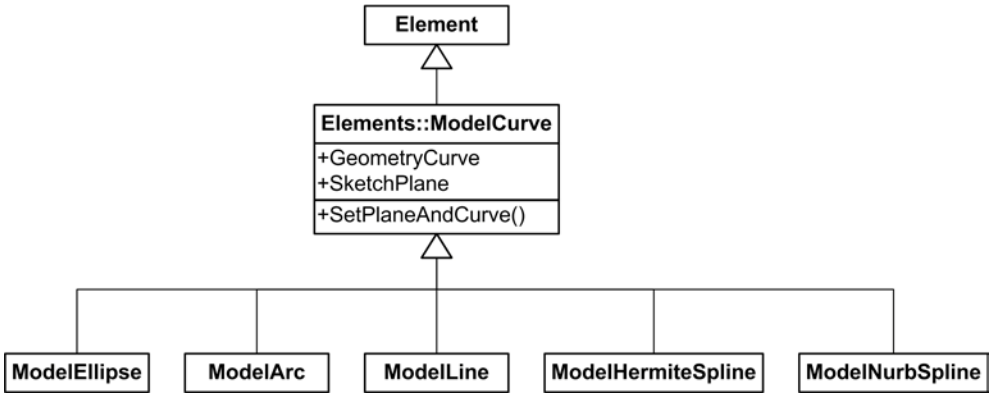



**Figure 127: ModelCurve diagram**

The following pictures illustrate the four ModelCurve derived classes:

**Figure 128:ModelLine and ModelArc**

**Figure 129: ModelEllipse and ModelNurbSpline**

### 13.3.1 Create

The key to creating a ModelCurve is to create the Geometry.Curve and SketchPlane where the Curve is located. Based on the Geometry.Curve type you input, the corresponding ModelCurve returned can be downcast to its correct type.

The following sample illustrates how to create a new model curve (ModelLine and ModelArc):

**Code Region 2-Sketching**

```
//… other operation
Autodesk.Revit.Application app = commandData.Application;
Document doc = app.ActiveDocument;

// Create a geometry line in revit application
XYZ startPoint = new XYZ(0, 0, 0);
XYZ endPoint = new XYZ(10, 10, 0);
Line geomLine = app.Create.NewLine(ref startPoint, ref endPoint, true);
```

```
// Create a geometry arc in revit application
XYZ end0 = new XYZ(1, 0, 0);
XYZ end1 = new XYZ(10, 10, 10);
XYZ pointOnCurve = new XYZ(10, 0, 0);
Arc geomArc = app.Create.NewArc(ref end0, ref end1, ref pointOnCurve);

// Create a geometry plane in revit application
XYZ origin = new XYZ(0, 0, 0);
XYZ normal = new XYZ(1, 1, 0);
Plane geomPlane = app.Create.NewPlane(ref normal, ref origin);

// Create a sketch plane in current document
SketchPlane sketch = doc.Create.NewSketchPlane(geomPlane);

// Create a ModelLine element using the created geometry line
// and sketch plane
ModelLine line = doc.Create.NewModelCurve(geomLine, sketch) as ModelLine;

// Create a ModelArc element using the created geometry arc
// and sketch plane
ModelArc arc = doc.Create.NewModelCurve(geomArc, sketch) as ModelArc;

//…other operation
```

### 13.3.2 Properties

ModelCurve has properties that help you set specific GeometryCurves. In this section, the GeometryCurve and LineStyle properties are introduced.

#### 13.3.2.1 GeometryCurve

The GeometryCurve property is used to get or set the model curve's geometry curve. Except for ModelHermiteSpline, you can get different Geometry.Curves from the four ModelCurves;

- Line
- Arc
- Ellipse
- Nurbspline.

The following code sample illustrates how to get a specific Curve from a ModelCurve.

**Code Region 3-Sketching**

```
//get the geometry curve of the model curve
Autodesk.Revit.Geometry.Line geoCurve =
modelCurve.GeometryCurve as Autodesk.Revit.Geometry.Line;
```

The GeometryCurve property return value is a general Geometry.Curve object, therefore, you must use an As operator to convert the object type. For more details about using the As operator, see the As and Is Operators section in the Collection chapter or reference the related C# material.

**Note:** The following information applies to GeometryCurve:

- In Revit you cannot create a Hermite curve but you can import it from other software such as AutoCAD. Geometry.Curve is the only geometry class that represents the Hermite curve.
- The SetPlaneAndCurve method and the Curve and SketchPlane property setters are used in different situations.
  - When the new Curve lies in the same SketchPlane, or the new SketchPlane lies on the same planar face with the old SketchPlane, use the Curve or SketchPlane property setters.
  - If new Curve does not lay in the same SketchPlane, or the new SketchPlane does not lay on the same planar face with the old SketchPlane, you must simultaneously change the Curve value and the SketchPlane value using SetPlaneAndCurve to avoid internal data inconsistency.

#### 13.3.2.2 LineStyle

Currently line style does not have a specific class but is represented by the Document.Element class.

- All line styles for a ModelCurve are available from the LineStyles property in the Element Properties dialog box.
- The Element object that represents line style cannot provide information for all properties. Most properties will return null or an empty collection.
- The only information returned is the following:
  - Id
  - UniqueId
  - Name.
- Use the getLineStyle method to retrieve the current line style or use the setLineStyle method to set the current line style to one returned from the LineStyles property.

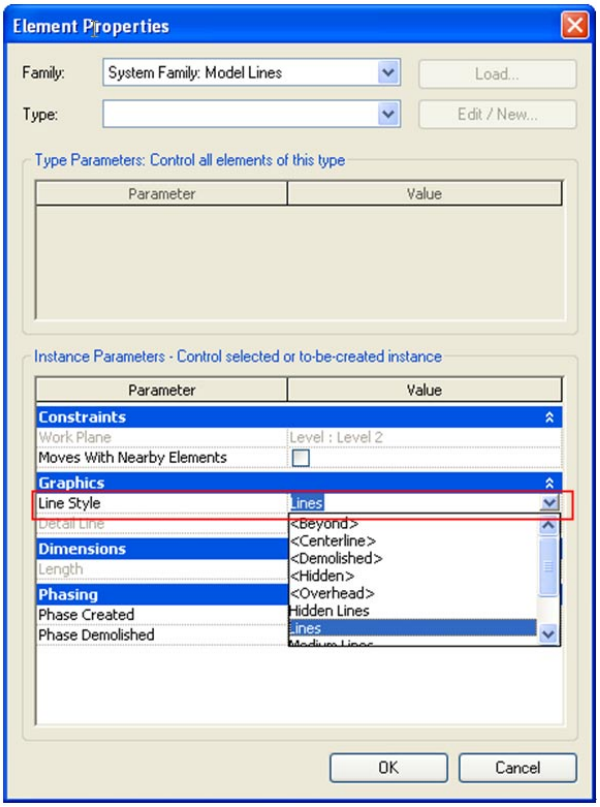



**Figure 130: Line Style parameter**

# 14 Views

Views are images produced from a Revit model with privileged access to the data stored in the documents. They can be graphics, such as plans, or text, such as schedules. Each project document has one or more different views. The last focused window is the active view.

In this chapter, you learn how views are generated, the types of views supported by Revit, and the features for each view.

## 14.1 Overview

This section is a high-level overview discussing the following:

- How views are generated.
- View types.
- Element visibility.
- Create and delete views.

### 14.1.1 View Process

The following figure illustrates how a view is generated.

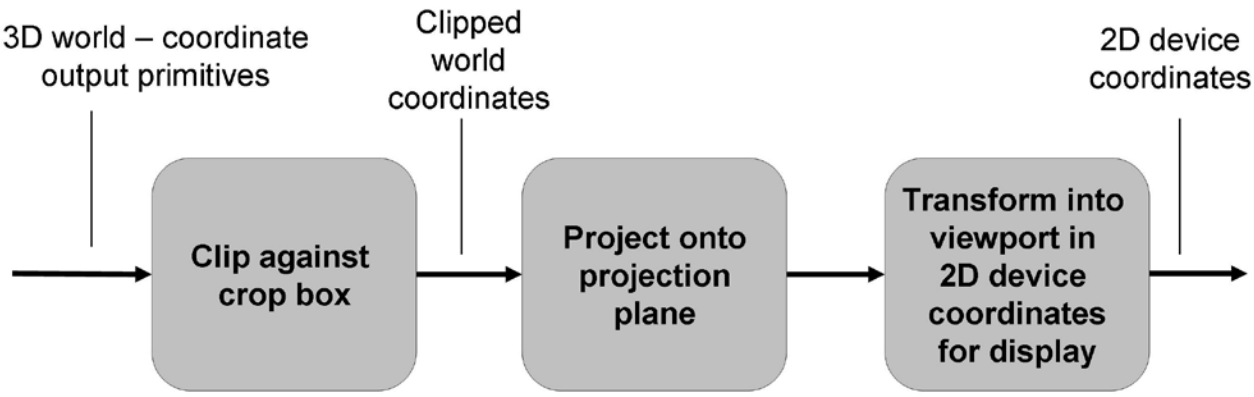

3D world – coordinate output primitives
Clipped world coordinates
2D device coordinates
Clip against crop box
Project onto projection plane
Transform into viewport in 2D device coordinates for display

**Figure 131: Create view process**

Each view is generated by projecting a three-dimensional object onto a two-dimensional projection plane. Projections are divided into two basic classes:

- Perspective
- Parallel

After the projection type is determined, you must specify the conditions under which the 3D model is needed and the scene is to be rendered. For more information about projection, refer to the View3D section.

World coordinates include the following:

- The viewer's eye position
- The viewing plane location where the projection is displayed.

Revit uses two coordinate systems

- The global or model space coordinates where the building exists
- The viewing coordinate system.

The viewing coordinate system represents how the model is presented in the observer's view. Its origin is the viewer's eye position whose coordinates in the model space are retrieved by the View.Origin property. The X, Y, and Z axes are represented by the View.RightDirection, View.UpDirection, and View.ViewDirection properties respectively.

- View.RightDirection is towards the right side of the screen.
- View.UpDirection towards the up side of the screen.
- View.ViewDirection from the screen to the viewer.

The viewing coordinate system is right-handed. For more information, see the Perspective Projection picture and the Parallel Projection picture in the View3D section in this chapter.

Some portions of a 3D model space that do not display, such as those that are behind the viewer or too far away to display clearly, are excluded before being projected onto the projection plane. This action requires cropping the view. The following rules apply to cropping:

- Elements outside of the crop box are no longer in the view.
- The View.CropBox property provides the geometry information for the box. It returns an instance of BoundingBoxXYZ indicating a rectangular parallelepiped in the viewing coordinate system. The coordinate system and the crop box shape are illustrated in the View3D section.

After cropping, the model is projected onto the projection plane. The following rules apply to the projection:

- The projection contents are mapped to the screen view port for display.
- During the mapping process, the projection contents are scaled so that they are shown properly on the screen.
  - The View.Scale property is the ratio of the actual model size to the view size.
  - The view boundary on paper is the crop region, which is a projection of the crop box on the projection plane.
  - The size and position of the crop region is determined by the View.OutLine property.

### 14.1.2 View Types

A project model can have several view types. The following picture demonstrates the different types of views in the Project browser.

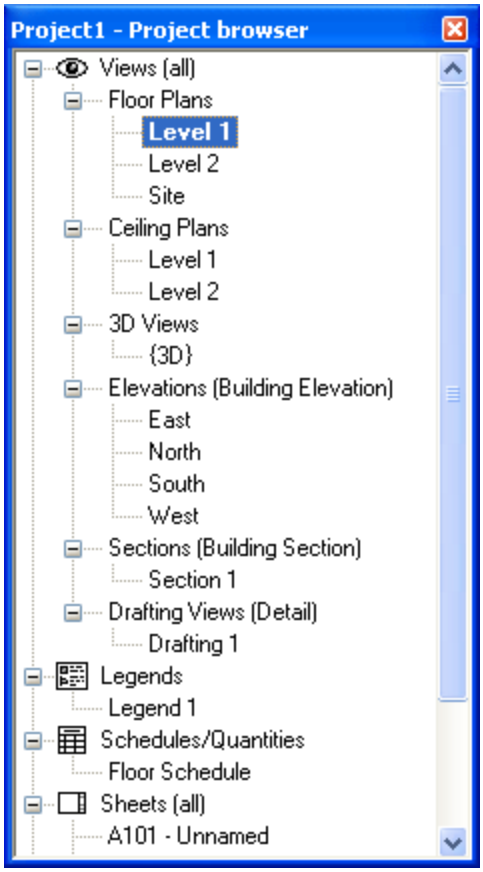



**Figure 132: Different views in the Project browser**

In the API, there are two ways to classify all views. The first way is by using the view element View.ViewType property. It returns an enumerated value indicating the view type. The following table lists all available view types.

| Member Name | Description |
|---|---|
| AreaPlan | Area view. |
| CeilingPlan | Reflected ceiling plan view. |
| ColumnSchedule | Coulmn schedule view. |
| CostReport | Cost report view. |
| Detail | Detail view. |
| DraftingView | Drafting view. |
| DrawingSheet | Drawing sheet view. |
| Elevation | Elevation view. |
| EngineeringPlan | Engineering view. |
| FloorPlan | Floor plan view. |
| Internal | Revit's internal view. |
| Legend | Legend view. |
| LoadsReport | Loads report view. |
| PanelSchedule | Panel schedule view. |
| PresureLossReport | Presure Loss Report view. |

| Member Name | Description |
|---|---|
| Rendering | Rendering view. |
| Report | Report view. |
| Schedule | Schedule view. |
| Section | Cross section view. |
| ThreeD | 3-D view. |
| Undefined | Undefined/unspecified view. |
| Walkthrough | Walkthrough view. |

**Table 36: Autodesk.Revit.Enums.ViewTypes**

The second way to classify views is by the class type. The following figure illustrates the view class hierarchy in the Revit API.

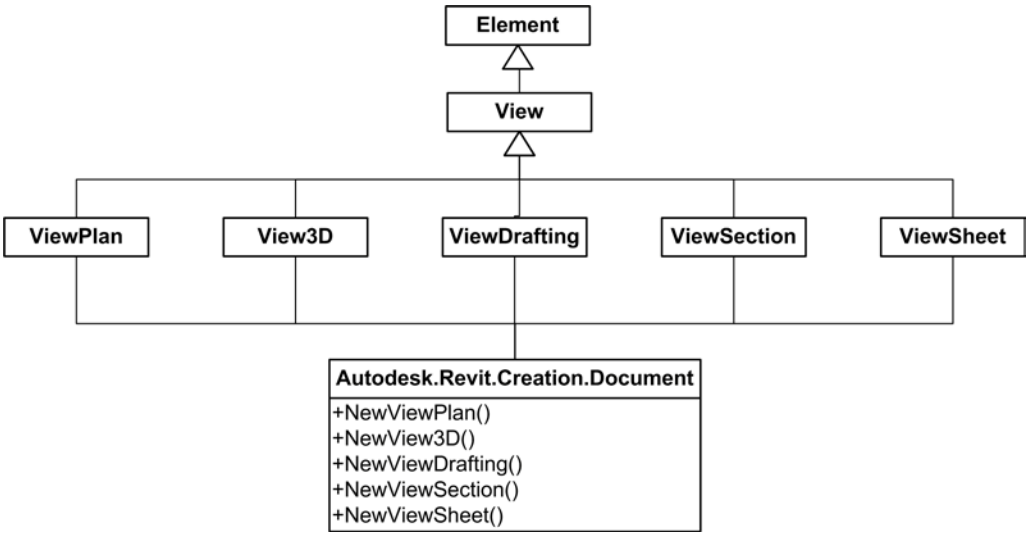



**Figure 133: View classes in the Revit API diagram**

The following table lists the view types and the corresponding views in the Project browser.

| Project Browser Views | View Type | Class Type |
|---|---|---|
| Area Plans | ViewType.AreaPlan | Elements.ViewPlan |
| Ceiling Plans | ViewType.CeilingPlan | Elements.ViewPlan |
| Graphic Column Schedule | ViewType.ColumnSchedule | Elements.View |
| Reports | ViewType.CostReport | Elements.View |
| Detail Views | ViewType.Detail | Elements.ViewSection |
| Drafting Views | ViewType.DraftingView | Elements.ViewDrafting |
| Sheets | ViewType.DrawingSheet | Elements.ViewSheet |
| Elevations | ViewType.Elevation | Elements.View |
| Structural Plans (Revit Structure) | ViewType.EngineeringPlan | Elements.ViewPlan |
| Floor Plans | ViewType.FloorPlan | Elements.ViewPlan |

| Project Browser Views | View Type | Class Type |
|---|---|---|
| Project Browser, System Navigator (Unavailable to API user) | ViewType.Internal | Elements.View |
| Legends | ViewType.Legend | Elements.View |
| Reports (Revit MEP) | ViewType.LoadsReport | Elements.View |
| Reports (Revit MEP) | ViewType.PanelSchedule | Elements.View |
| Reports (Revit MEP) | ViewType.PresureLossReport | Elements.View |
| Renderings | ViewType.Rendering | Elements.ViewDrafting |
| Reports | ViewType.Report | Elements.View |
| Schedules/Quantities | ViewType.Schedule | Elements.View |
| Sections | ViewType.Section | Elements.View |
| 3D Views | ViewType.ThreeD | Elements.View3D |
| ---- | ViewType.Undefined | ---- |
| Walkthroughs | ViewType.Walkthrough | Elements.View3D |

**Table 37: Project Browser Views**

In the following example, if variable v is an Elements.View object, you can get the variable view type and class type, using the following code.

**Code Region 1-View**

```
//Autodesk.Revit.Elements.View view = GetView();
//Two ways to judge the type of a view:
//first way:
ViewType viewType = view.ViewType;
switch (viewType)
{
   case Autodesk.Revit.Enums.ViewType.ThreeD:
      // View type of v is ThreeD
      break;
   // Get other types
}

// second way:
if (view is View3D)
{
   // Class type of v is View3D
}
```

### 14.1.3 Element Visibility in a View

Views keep track of visible elements. All elements that are graphical and visible in the view can be retrieved using the View.Elements property. This property returns a set of elements visible in the

view. However, some elements in the set may be hidden or covered by other elements. You can see them by rotating the view or removing the elements that cover them.

Elements are shown or hidden in a view by category.

- The View.getVisiblility method queries a category to determine if it is visible or invisible in the view.
- The View.setVisibility method sets all elements in a specific category to visible or invisible.

The set returned by View.Elements only contains elements visible in the current view. You cannot retrieve elements that are not graphical or elements that are invisible. Document.Elements retrieves all elements in the document including invisible elements and non-graphical elements. For example, when creating a default 3D view in an empty project, there are no elements in the view but there are many elements in the document, all of which are invisible.

**Note:** Autodesk.Revit.Document also has a property named Elements. It is important to distinguish the Autodesk.Revit.Elements.View.Elements property from the Autodesk.Revit.Document.Elements property based on the previous description and the following code sample.

The following code sample counts the number of elements in the active document and active view. The number of elements in the active view changes if you hide some elements while the number of elements in the document is constant.

**Code Region 2-View**

```
private void CountElements(Autodesk.Revit.Document doc)
{
   // Count the number of elements in the document
   int count = 0;
   ElementIterator iter = doc.Elements;
   iter.Reset();
   while (iter.MoveNext())
   {
      Autodesk.Revit.Element elem = iter.Current as Autodesk.Revit.Element;
      if (null != elem)
         ++count;
   }

   StringBuilder message = new StringBuilder();
   message.AppendLine("Elements within Document: " + count.ToString());
   message.AppendLine("Elements within active View: "
      + doc.ActiveView.Elements.Size.ToString());

   MessageBox.Show(message.ToString());
}
```

### 14.1.4 Create and Delete Views

The Revit API provides five methods to create the corresponding view elements derived from Autodesk.Revit.Elements.View class.

| Method | Parameters |
|---|---|

| Method | Parameters |
|---|---|
| View3D NewView3D( ref XYZ viewDirection) | viewDirection:<br>Vector pointing towards the viewer's eye. |
| ViewPlan NewViewPlan(<br>string pViewName,<br>Level pLevel,<br>ViewType viewType<br>) | pViewName:<br>Name for the new plan view. It must be unique or a null pointer.<br>pLevel:<br>Level associated with the plan view.<br>viewType:<br>Type of plan view created. It must be Floor Plan or Ceiling Plan (Structural Plan in Structure). |
| ViewSection NewViewSection(<br>BoundingBoxXYZ box<br>) | box:<br>View orientation and bounds. The X axis points towards the right of screen; Y - towards up; Z - towards the user. |
| ViewSheet NewViewSheet(<br>FamilySymbol titleBlock<br>) | titleBlock:<br>The titleblock family symbol applied to the sheet. |
| ViewDrafting NewViewDrafting(<br>) | |

**Table 38: Create View Element Methods**

If a view is created successfully, these methods return a reference to the view, otherwise it returns null. The methods are described in the following sections.

Delete a view by useing the Document.Delete method with the view ID. You can also delete view-dependent elements. For example, deleting the level element causes Revit to delete the corresponding plan view or deleting the camera element causes Revit to delete the corresponding 3D view.

**Note:** Currently, you cannot gain access to Schedule views in the Revit 2008 API as there is no NewScheduleView method.

## 14.2 View3D

View3D is a freely-oriented three-dimensional view. There are two kinds of 3D views, perspective view and orthographic view. The difference between the two views is based on the projection ray relationship. The View3D.IsPerspective property indicates whether a 3D view is perspective or orthographic.

Complete the following steps to add a View3D by adding a camera.

1. From the toolbar click View > New > Camera.
2. Check the Perspective check box to add a perspective view or leave it blank to add an orthographic view.
3. Click the drawing area to place the camera.
4. Select a point to specify the sight direction.

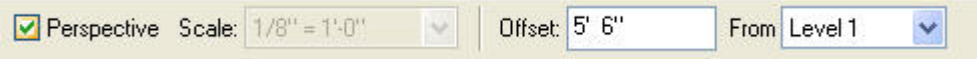



**Figure 134: Perspective check box**

### 14.2.1 Perspective View

The following picture illustrates how a perspective view is created.

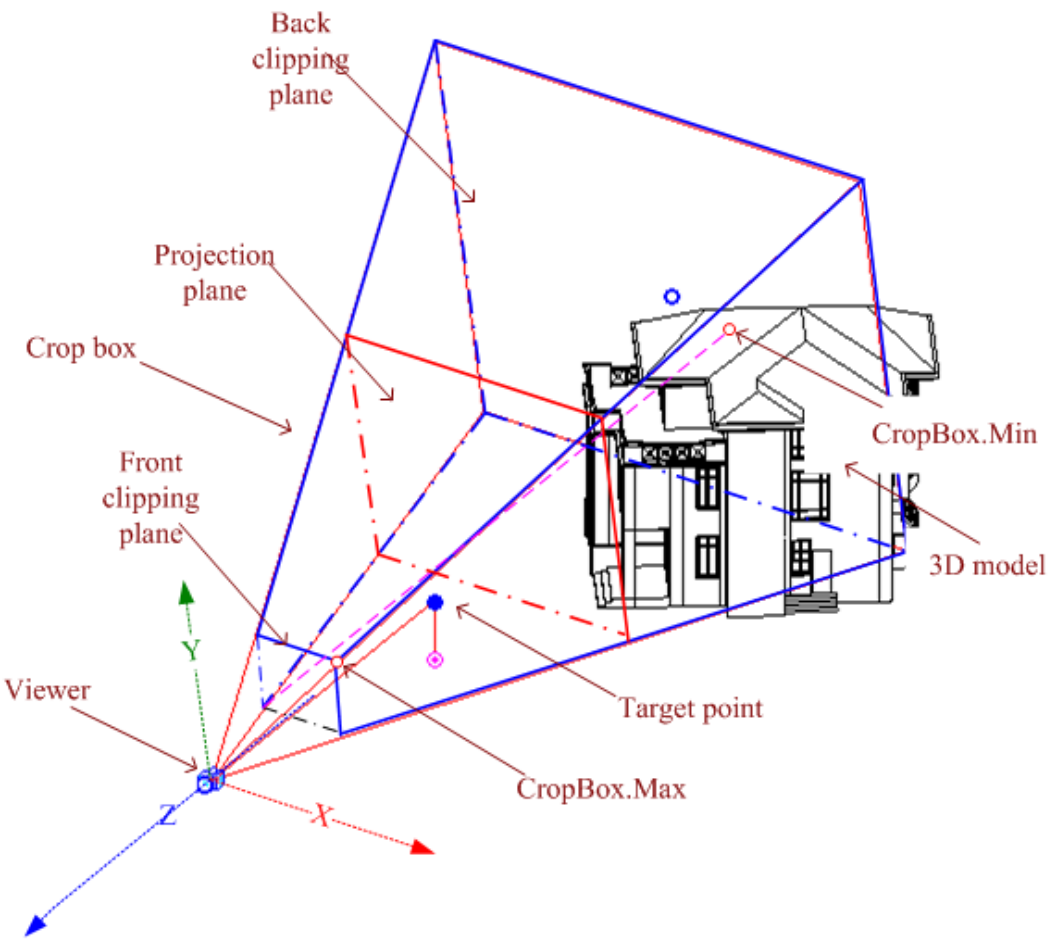



**Figure 135: Perspective projection**

- The straight projection rays pass through each point in the model and intersect the projection plane to form the projection contents.
- To facilitate the transformation from the world coordinate onto the view plane, the viewing coordinate system is based on the viewer.
- Its origin, represented by the View.Origin property, is the viewer position.
- The viewer's world coordinates are retrieved usng the View3D.EyePosition property. Therefore, in 3D views, View.Origin is always equal to View3D.EyePosition.
- As described, the viewing coordinate system is determined as follows:
  - The X-axis is determined by View.RightDirection.
  - The Y-axis is determined by View.UpDirection.
  - The Z-axis is determined by View.ViewDirection.
- The view direction is from the target point to the viewer in the 3D space, and from the screen to the viewer in the screen space.

The perspective view crop box is part of a pyramid with the apex at the viewer position. It is the geometry between the two parallel clip planes. The crop box bounds the portion of the model that is clipped out and projected onto the view plane.

- The crop box is represented by the View.CropBox property, which returns a BoundingBoxXYZ object.
- The CropBox.Min and CropBox.Max points are marked in the previous picture. Note that the CropBox.Min point in a perspective view is generated by projecting the crop box front clip plane onto the back clip plane.

Crop box coordinates are based on the viewing coordinate system. Use Transform.OfPoint to transform CropBox.Min and CropBox.Max to the world coordinate system. For more detail about Transform, refer to the Geometry.Transform section in the Geometry chapter.

Project plane plus the front and back clip plane are all plumb to the view direction. The line between CropBox.Max and CropBox.Min is parallel to the view direction. With these factors, the crop box geometry can be calculated.

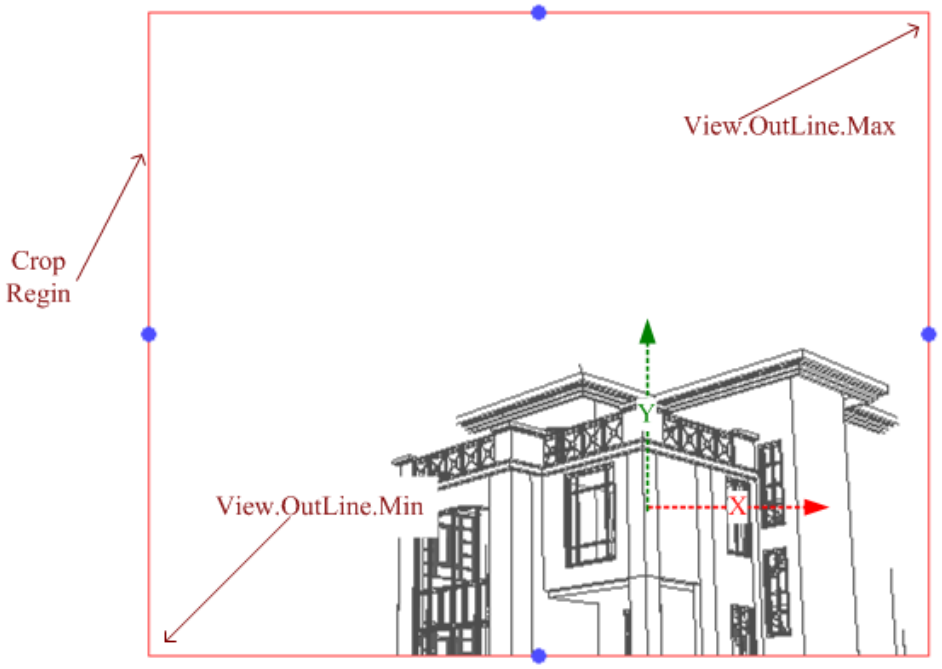



**Figure 136: Perspective 3D view**

The previous picture shows the projection plane on screen after cropping. The crop region is the rectangle intersection of the projection plane and crop box.

- Geometry information is retrieved using the View.CropRegion property. This property returns a BoundingBoxUV instance.
- The View.OutLine.Max property points to the upper right corner.
- The View.OutLine.Min property points to the lower left corner.

Like the crop box, the crop region coordinates are based on the viewing coordinate system. The following expressions are equal.

View.CropBox.Max.X(Y) / View.OutLine.Max.X(Y)

== View.CropBox.Min.X(Y) / View.OutLine.Min.X(Y)

Since the size of an object's perspective projection varies inversely with the distance from that object to the center of the projection, scale is meaningless for perspective views. The perspective 3D view Scale property always returns zero. Currently, the API cannot create Perspective views.

### 14.2.2 Orthographic View

Autodesk.Revit.Creation.Document provides the NewView3D method to create an orthographic 3D view.

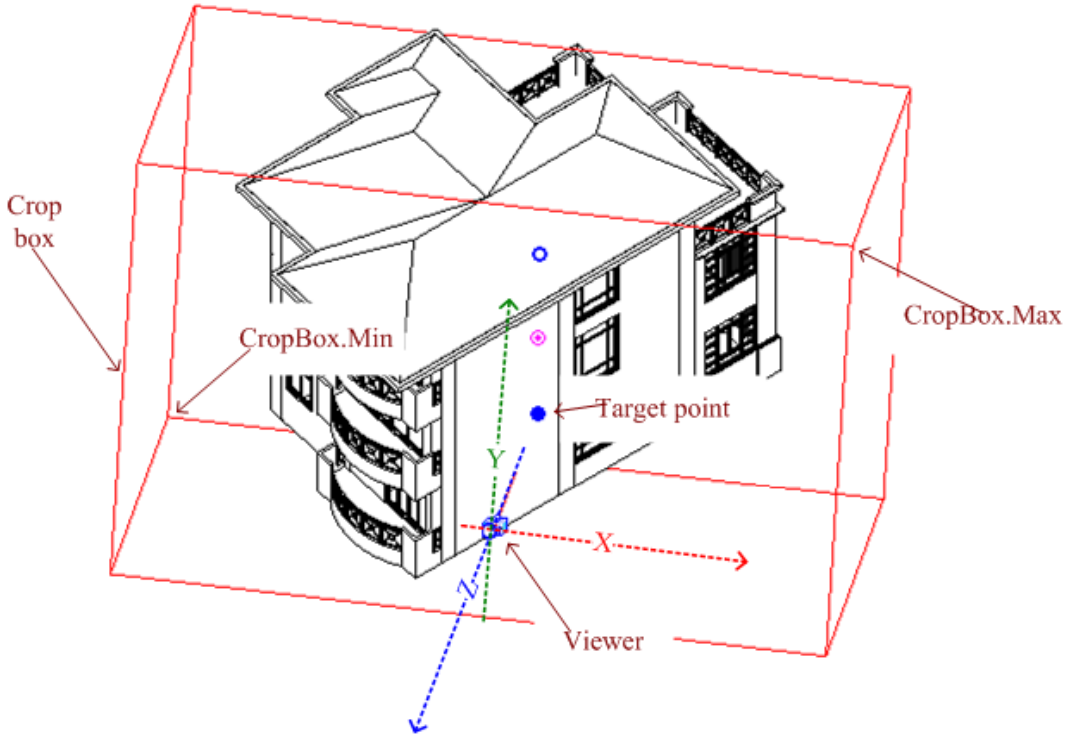



**Figure 137: Parallel projection**

Orthographic views are generated using parallel projection rays by projecting the model onto a plane that is normal to the rays. The viewing coordinate system is similar to the perspective view, but the crop box is a parallelepiped with faces that are parallel or normal to the projection rays. The View.CropBox property points to two diagonal corners whose coordinates are based on the viewing coordinate system.

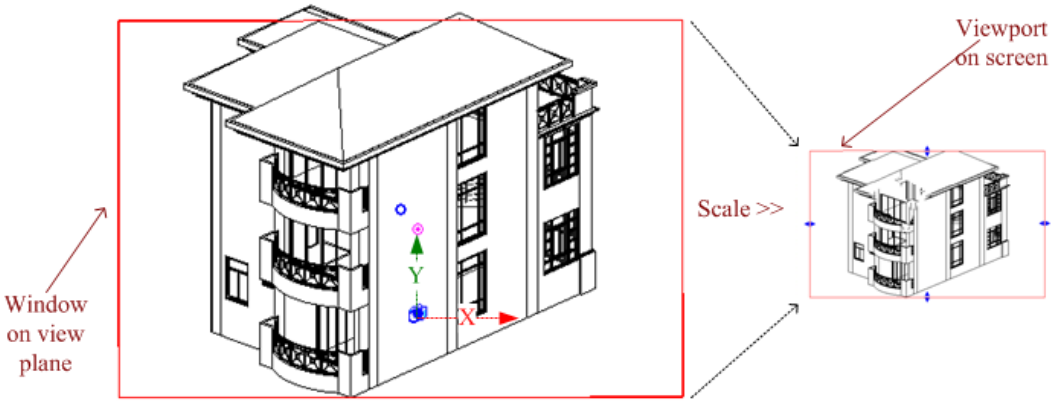



**Figure 138: Scale the window on view plane to screen viewport**

The model is projected onto a view plane and then scaled onto the screen. The View.Scale property represents the ratio of actual model size to the view size. The related expressions are as follows:

View.CropBox.Max.X(Y) / View.OutLine.Max.X(Y)

== View.CropBox.Min.X(Y) / View.OutLine.Min.X(Y)

== View.Scale

```
public View3D NewView3D(ref XYZ viewDirection);
```

Create an orthographic 3D view by passing the view vector to the Autodesk.Revit.Creation.Document.NewView3D method. This vector can be a unit vector or not. Revit determines the following:

- Position of the viewer.
- How to create the viewing coordinate system using the view direction.
- How to create the crop box to crop the model.

Once the view is created, you can resize the crop box to view different portions of the model. Currently, the API does not support modifying the viewing coordinate system.

The following code sample illustrates how to create a 3D view. The created view is orthographic.

**Code Region 3-View**

```
private void CreateView3D(Autodesk.Revit.Document doc)
{
   try
   {
      // Create a new View3D
      XYZ direction = new XYZ(1, 1, 1);
      View3D view3D = doc.Create.NewView3D(ref direction);
      if (null == view3D)
         return;

      // The created View3D isn't perspective.
      Debug.Assert(false == view3D.IsPerspective);
   }
   catch (Exception e)
   {
      Debug.WriteLine(e.ToString());
   }
}
```

### 14.2.3 3D Views SectionBox

Each view has a crop box. The crop box focuses on a portion of the model to project and show in the view. For 3D views, there is another box named section box.

- The section box determines which model portion appears in a 3D view.
- The section box is used to clip the 3D model's visible portion.

- The part outside the box is invisible even if it is in the crop box.
- The section box is different from the crop box in that it can be rotated and moved with the model.

The section box is particularly useful for large models. For example, if you want to render a large building, use a section box. The section box limits the model portion used for Radiosity or Raytrace calculation. To display the section box, in the 3D view Element Properties dialog box, select Section Box in the Extents section. You can also set it using the API:

**Code Region 4-View**

```
private void ShowHideSectionBox(Autodesk.Revit.Elements.View3D view3D)
{
   foreach (Parameter p in view3D.Parameters)
   {
      // Get Section Box parameter
     if (p.Definition.Name.Equals("Section Box"))
     {
         // Show Section Box
        p.Set(1);
         // Hide Section Box
        // p.Set(0);
        break;
     }
   }
}
```

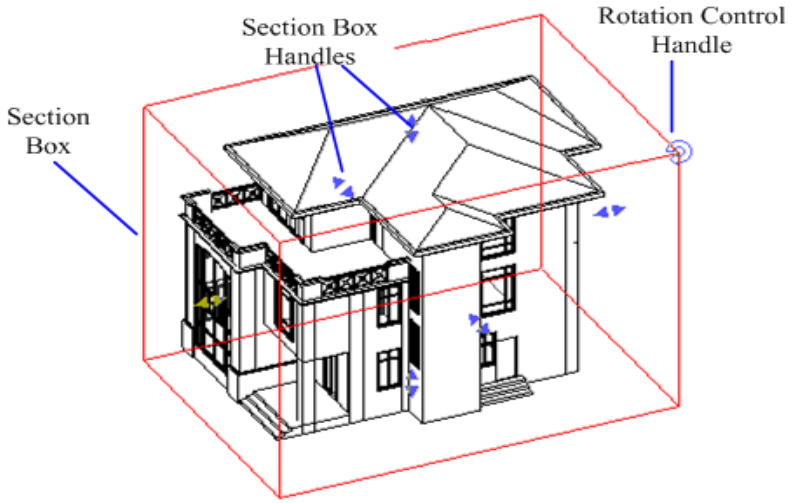



**Figure 139: Section box**

The View3D.SectionBox property is used to get and change the box extents. In some cases, setting the View3D.SectionBox can have a side effect. Setting the property to certain values can change the box capacity and display it in the view. However, you can assign a null value to the SectionBox to keep the modified value and make the section box invisible in the view. To avoid displaying the section box, change the section box value, then set the section box to null. The following code sample illustrates this process. Notice it only works when the Section Box check box is selected in the View property dialog box.

**Code Region 5-View**

```
private void ExpandSectionBox(Autodesk.Revit.Elements.View3D view)
{
   // The orignial section box
   BoundingBoxXYZ sectionBox = view.SectionBox;

   // Expand the section box
   XYZ deltaXYZ = sectionBox.Max - sectionBox.Min;
   sectionBox.Max += deltaXYZ / 2;
   sectionBox.Min -= deltaXYZ / 2;

   //After reseting the section box, it will be shown in the view.
  //It only works when the Section Box check box is
  //checked in View property dialog.
   view.SectionBox = sectionBox;

  //Setting the section box to null will make it hidden.
   view.SectionBox = null;       // line x
}
```

**Note:** If you set view.SectionBox to null, it has the same effect as hiding the section box using the Section Box parameter. The current section box is stored by view and is restored when you show the section box using the SectionBox parameter.

## 14.3 ViewPlan

Plan views are level-based. There are two types of plan views, floor plan view and ceiling plan view.

- Generally the floor plan view is the default view opened in a new project.
- Most projects include at least one floor plan view and one ceiling plan view.
- Plan views are usually created after adding new levels to the project.

Adding new levels using the API does not add plan views automatically. Autodesk.Revit.Creation.Document provides a NewViewPlan method to create a plan view.

**Code Region 5-View**

```
public ViewPlan NewViewPlan(string pViewName, Level pLevel, ViewType viewType);
```

The viewType parameter must be FloorPlan or CeilingPlan. The level parameter represents a level element in the project to which the plan view is associated.

The following code creates a floor plan and a ceiling plan based on a certain level.

**Code Region 6-View**

```
private void CreatePlanView(Autodesk.Revit.Document doc)
{
   try
   {
```

```
      // Create a Level and a Floor Plan based on it
      double elevation = 10.0;
      Level level1 = doc.Create.NewLevel(elevation);
      ViewPlan floorView = doc.Create.NewViewPlan(null, level1, ViewType.FloorPlan);

      // Create another Level and a Ceiling Plan based on it
      elevation += 10.0;
      Level level2 = doc.Create.NewLevel(elevation);
      ViewPlan ceilingView = doc.Create.NewViewPlan(null, level2, ViewType.CeilingPlan);
   }
   catch (Exception exp)
   {
      MessageBox.Show(exp.ToString());
   }
}
```

## 14.4 ViewDrafting

The drafting view is not associated with the model. It allows the user to create detail drawings that are not included in the model.

- In the drafting view, the user can create details in different view scales (coarse, fine, or medium).
- You can use 2D detailing tools, including:
  - Detail lines
  - Detail regions
  - Detail components
  - Insulation
  - Reference planes
  - Dimensions
  - Symbols
  - Text

  These tools are the same tools used to create a detail view.
- Drafting views do not display model elements.

Use the Autodesk.Revit.Creation.NewViewDrafting method to create a drafting view. Model elements are not displayed in the drafting view.

## 14.5 ViewSection

Section views cut through the model to expose the interior structure. The Auotdesk.Revit.Creation.NewViewSection method creates the section view.

```
public ViewSection NewViewSection(BoundingBoxXYZ box);
```

The box parameter is the section view crop box. It provides the orientation and bounds which are required for the section view. Usually, another view's crop box is used as the parameter. You can also build a custom BoundingBoxXYZ instance to represent the orientation and bounds.

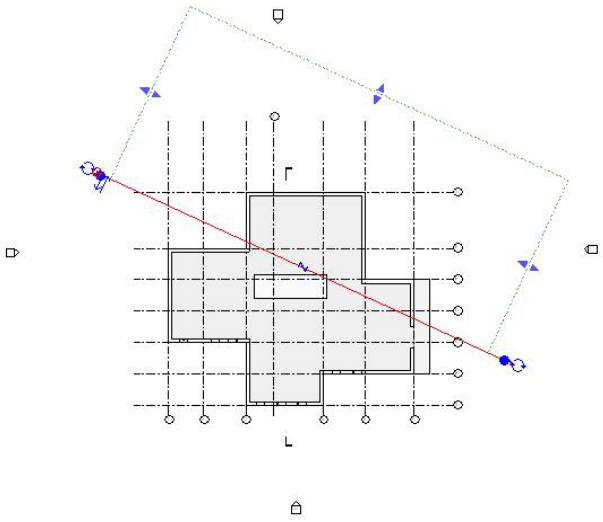


**Figure 140: Plan view section**

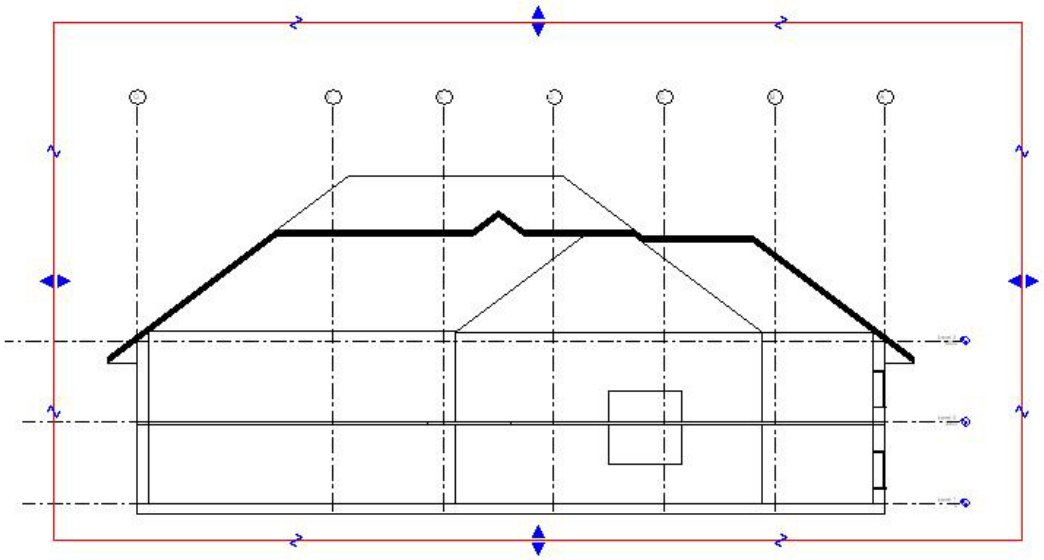


**Figure 141: Section view section**

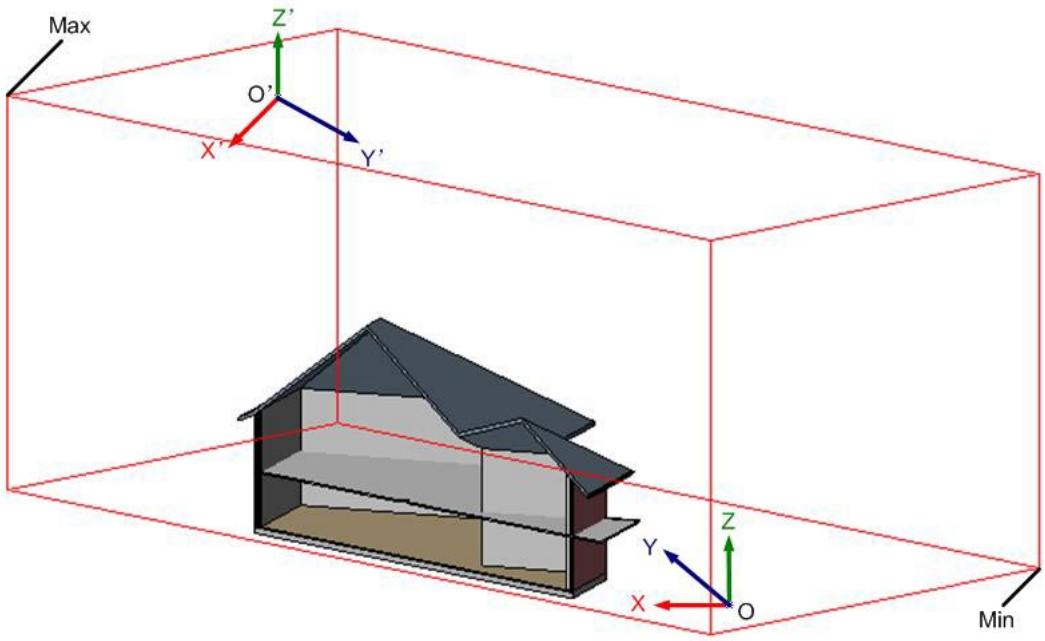



**Figure 142: Section view box**

For example, apply the box to the section view create method. The coordinate system O-XYZ is the model coordinate system and the box Max and Min points are based on this system. The created section view introduces a new system, O`-X`Y`Z`, as the viewing coordinate system.

- Point O` is the section view origin.
- Its coordinates are retrieved using the View.Origin property.
- The section view crop box is calculated using the box.

**Note:** When you create a section using the command View > New > Section, the created view is located in the Sections (Building Section) node in the Project Browser. The view class type is Elements.View, and the view type is ViewType.Section. However, when you create a section using the API Autodesk.Revit.Creation.NewViewSection method, the created view is in the Detail Views (Detail) node. The class type is Elements.ViewSection and the view type is ViewType.Detail.

## 14.6 ViewSheet

A sheet is a view used to prepare a finished printout. It contains views and a title block. When creating a sheet view with the Autodesk.Revit.Creation.NewViewSheet method, a title block family symbol is a required parameter for the method. The Autodesk.Revit.Document TitleBlocks property contains all title blocks in the document. Choose one title block to create the sheet.

```
public void AddView(View newView, ref UV location);
```

The newly created sheet has no views. The ViewSheet.AddView method is used to add views.

- The ref Geometry.UV location parameter identifies where the added views are located. It points to the added view's center coordinate (measured in inches).
- The coordinates, [0, 0], are relative to the sheet's lower left corner.

Each sheet has a unique sheet number in the complete drawing set. The number is displayed before the sheet name in the Project Browser. It is convenient to use the sheet number in a view

title to cross-reference the sheets in your drawing set. You can retrieve or modify the number using the SheetNumber property. The number must be unique; otherwise an exception is thrown when you set the number to a duplicate value.

The following example illustrates how to create and print a sheet view. Begin by finding an available title block in the document and use it to create the sheet view. Next, add the document active view. The active view is placed in the center of the sheet. Finally, print the sheet by calling the View.Print method.

**Code Region 7-View**

```
private void CreateSheetView(Autodesk.Revit.Document doc)
{
   // Get an available title block from document
   FamilySymbolSet fsSet = doc.TitleBlocks;
   if (fsSet.Size == 0)
   {
      MessageBox.Show("No title blocks");
      return;
   }

   FamilySymbol fs = null;
   foreach (FamilySymbol f in fsSet)
   {
      if (null != f)
      {
         fs = f;
         break;
      }
   }

   try
   {
      // Create a sheet view
      ViewSheet viewSheet = doc.Create.NewViewSheet(fs);
      if (null == viewSheet)
         return;

      // Add current view onto the center of the sheet
      UV location = new UV(
        (viewSheet.Outline.Max.U - viewSheet.Outline.Min.U) / 2,
        (viewSheet.Outline.Max.V - viewSheet.Outline.Min.V)/ 2);
      viewSheet.AddView(doc.ActiveView, ref location);

      // Print the sheet out
      if (viewSheet.CanBePrinted)
      {
         if (MessageBox.Show("Print the sheet?", "Revit",
```

```
            MessageBoxButtons.YesNo) == DialogResult.Yes)
            viewSheet.Print();
        }
    }
    catch (Exception e)
    {
        MessageBox.Show(e.ToString());
    }
}
```

**Note:** You cannot add a sheet view to another sheet and you cannot add a view to more than one sheet; otherwise an argument exception occurs.

# 15 Material

In the Revit API, material data is stored and managed as an Element. The Material element includes Elements.Material and its subclasses:

- MaterialGeneric
- MaterialConcrete
- MaterialWood
- MaterialSteel
- MaterialOther.

Material features are represented by properties, such as FillPattern, Color, Render and so on.

Because Material is abstract, most functionality in the API is introduced by comparing the information with corresponding commands in the Revit UI.

In this chapter, you learn how to access material elements and how to manage the Material objects in the document. The section, Walkthrough: Get Window Materials, provides a walkthrough showing how to get a window material.

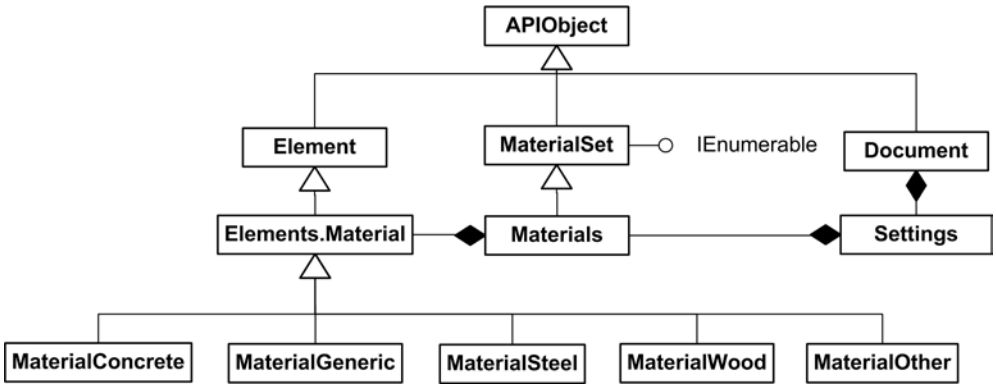



**Figure 143: Material-related classes diagram**

## 15.1 General Material Information

Before you begin the walkthrough, read through the following section for a better understanding of the Material class.

### 15.1.1 Classification

All Elements.Material objects are available in the Settings class Materials property. For more details, refer to the Management in Document section in this chapter. Material objects are also available in Document, Category, Element, Face, and so on, and are discussed in the pertinent sections in this chapter. Wherever you get a material object, it is represented as the Elements.Material class requiring you to downcast the object to its derived type.

There are two ways to downcast the Elements.Material object to its derived type. Use Runtime Type Information (RTTI) or BuiltInParameter.

#### 15.1.1.1 Use Runtime Type Information (RTTI)

The basic way to downcast the object is to use RTTI. For more information about the As operator, refer to the Collection chapter.

**Code Region 1- Material**

```
ElementIterator iter = document.Elements;
while (iter.MoveNext())
{
   MaterialSteel materialSteel = iter.Current as MaterialSteel;
   if (null != materialSteel)
   {
      //use it as a MaterialSteel object
   }
}
```

#### 15.1.1.2 Use BuiltInParameter

The second option is to use the BuiltInParameter PHY_MATERIAL_PARAM_TYPE. It is an Elements.Material class integer parameter. The parameter value identifies the Material type, as shown in the following table.

| Value | Type in API | Type in Revit |
|---|---|---|
| 0 | MaterialOther | Unassigned |
| 1 | MaterialConcrete | Concrete |
| 2 | MaterialSteel | Steel |
| 3 | MaterialGeneric | Generic |
| 4 | MaterialWood | Wood |

**Table 39: PHY_MATERIAL_PARAM_TYPE Parameter Values**

**Note:** Do not convert the integer parameter to Structural.Enums.Material. Structural.Enums.Material represents a structural FamilyInstance family parameter in the Revit Structure product. The FamilyInstance or FamilySymbol Material property is determined by the Family. It is not related to the Material object. In Revit, when editing a Structural Family (Settings > Family Category and Parameter), note that the Family Parameter, Structural Material Type, corresponds to the Structural.Enums.Material enumerated type.

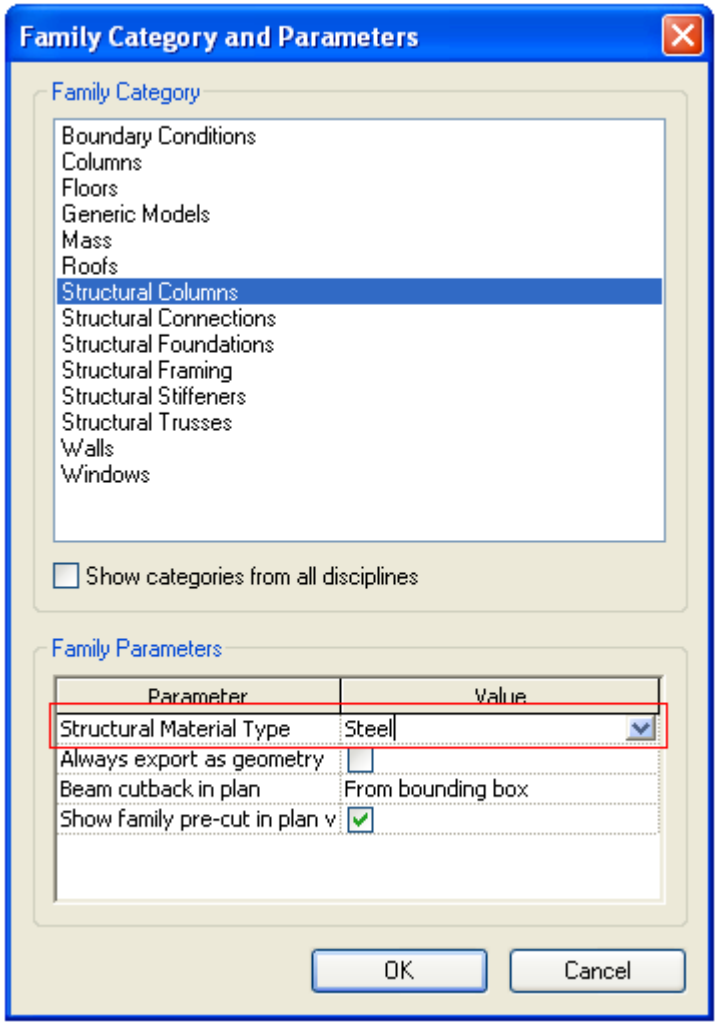



**Figure 144: Family parameter and Enum material**

Although there is no enumerated type for the Integer you can define one, as illustrated in the following code:

**Code Region 2- Material**

```
// Autodesk.Revit.Elements.Material material = GetMaterial();
Parameter parameter = material.get_Parameter(BuiltInParameter.PHY_MATERIAL_PARAM_TYPE);
int materialType = parameter.AsInteger();
if (2 == materialType)
{
   Debug.Assert(material is MaterialSteel);
   // use the MaterialSteel object...
}
```

There is no way to change the basic Material type. For example, you cannot change a MaterialSteel object to a MaterialWood object.

**Note:** Unlike the UI (shown below), the API does not provide access to the values of Concrete Type for Concrete material.

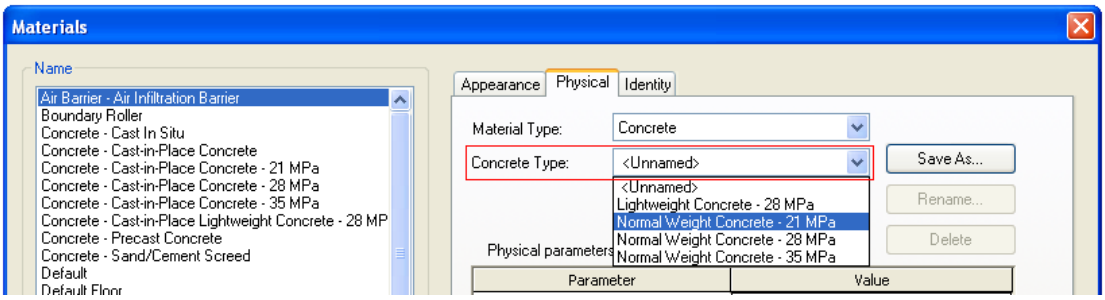



**Figure 145: Concrete Type Material**

## 15.1.2 Properties

The material object properties identify a specific type of material including color, fill pattern, and more.

### 15.1.2.1 Properties and Parameter

Elements.Material and its subclasses provide properties that have a counterpart Parameter. For example, the Color and Glow properties correspond to the BuiltInParameter MATERIAL_PARAM_COLOR and MATERIAL_PARAM_GLOW respectively.

In some cases, it is better to use a parameter rather than a property. For example, the AsValueString method (refer to the Parameter chapter sections AsValueString and SetValueString) is used to get a Parameter value in the unit. The result is converted to the same value as the one you see in Revit.

**Code Region 3- Material**

```
//Autodesk.Revit.Elements.MaterialSteel materialSteel = GetSteelMaterial();
//get young mod x
Parameter parameter =
  materialSteel.get_Parameter(BuiltInParameter.PHY_MATERIAL_PARAM_YOUNG_MOD1);
double dYOUNG_MOD = parameter.AsDouble();
string strYOUNG_MOD = parameter.AsValueString();
//YoungModulusX get by the two ways should be the same
Debug.Assert(dYOUNG_MOD == materialSteel.YoungModulusX);
```

### 15.1.2.2 Rendering Information

Currently, the AccuRender resource location is not available in Options.Application.LibraryPaths.

In the following Material Library dialog box picture, the API Material.RenderLibrary is _accurender, the Material.RenderName is Carpet\Solid Colors\Blue, Navy,Medium Pile. When setting the two Material properties, the API does not check to find out if the library or material texture exists. In addition, it does not create a new library if it does not exist.

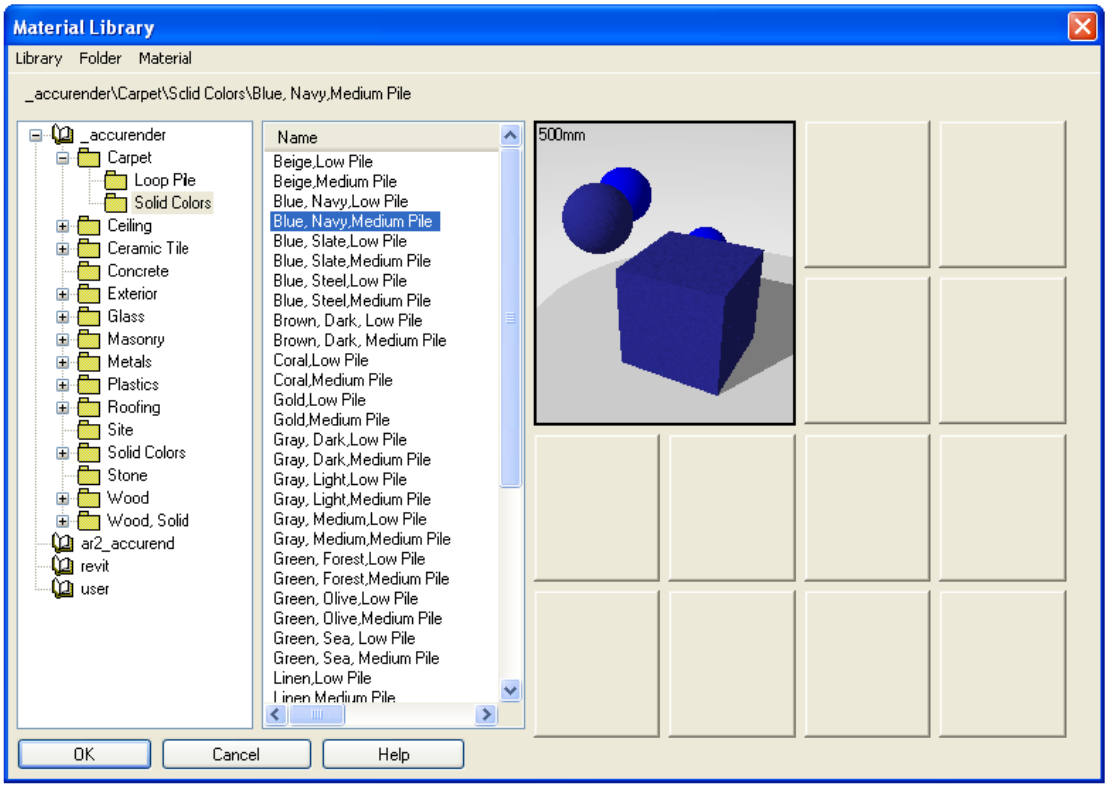



**Figure 146: RenderLibrary and RenderName**

The following sample shows how to use the Material RenderLibrary and RenderName properties:

**Code Region 4- Material**

```
//
// After executing the following code, the material Render texture property
// is changed to _accurender\Metals\Steel\Satin,Plain. You can
// view the change in UI.
//
material.RenderLibrary = "_accurender";
material.RenderName = "Metals\\Steel\\Satin,Plain";
```

### 15.1.2.3 FillPattern

All FillPatterns in a document are available in the Settings class FillPatterns property. Currently the FillPattern object only contains the Name and ID information. There are two kinds of FillPatterns: Drafting and Model. In the UI, you can only set Drafting fill patterns to Material.CutPattern. However, the classification is not exposed in the API. The following example shows how to change the material FillPattern.

**Code Region 5- Material**

```
foreach (FillPattern fillPattern in document.Settings.FillPatterns)
{
   // always set successfuly
   material.CutPattern = fillPattern;
   material.SurfacePattern = fillPattern;
   Debug.Assert(material.SurfacePattern.Name == fillPattern.Name);
```

```
   Debug.Assert(material.CutPattern.Name == fillPattern.Name);
}
```

## 15.2 Material Management

The MaterialSet object retrieved from the Settings class manages all Materials in the Document. Every Elements.Material object in the Document is identified by a unique name.

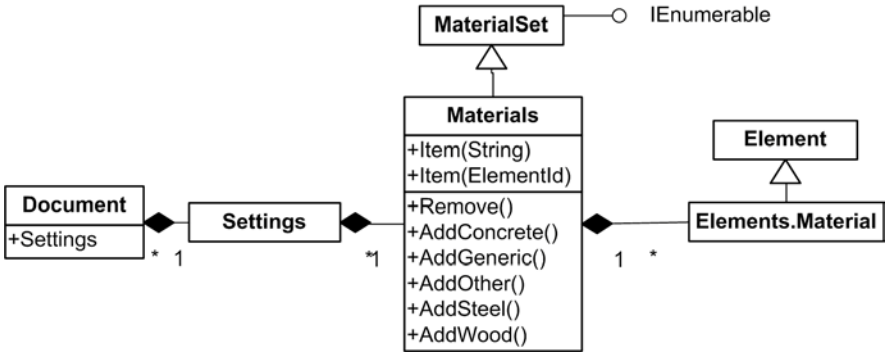



**Figure 147: Material management diagram**

The following example illustrates how to use the material name to get material and how to get the Document material set.

**Code Region 6- Material**

```
Materials materials = document.Settings.Materials;
Material floorMaterial = null;
string floorMaterialName = "Default Floor";
floorMaterial = materials.get_Item(floorMaterialName);
Debug.Assert(null != floorMaterial);

//or travel all the materials
foreach (Material material in materials)
{
   if (floorMaterialName == material.Name)
   {
      floorMaterial = material;
       break;
   }
}
Debug.Assert(null != floorMaterial);
```

**Note:** To run the sample code, make sure the material name exists in your document. All material names for the current document are located in the UI menu (Settings > Materials…).

### 15.2.1 Create

There are two ways to create a new Elements.Material object in the API.

- Duplicate an existing Material
- Add a new Material with a specific type.

When using the Duplicate method, the returned Material object has the same type as the original and it is added to the Materials collection automatically.

**Code Region 7- Material**

```
// Duplicate a material
string newMaterialName = "my new material";
if(!document.Settings.Materials.Contains(newMaterialName))
{
   MaterialSteel newMaterial = materialSteel.Duplicate(newMaterialName) as MaterialSteel;
   Debug.Assert(newMaterial.Name == newMaterialName);
   Debug.Assert(document.Settings.Materials.Contains(newMaterialName  ));
}
```

Use the Materials class to add a new Material directly. No matter how it is applied, it is necessary to specify a unique name. The unique name is the Elements.Material object key.

**Code Region 8- Material**

```
// Add a new material to document
Materials materials = document.Settings.Materials;
string materialName = "New Concrete Material";
if (!materials.Contains(materialName))
{
   materials.AddConcrete(materialName);
   Debug.Assert(materials.get_Item(materialName) is MaterialConcrete);
}
```

### 15.2.2 Delete

There are two ways to delete a material.

- Materials.Remove
- Document.Delete

The Materials.Remove method is specific to the Elements.Material class.

**Code Region 9- Material**

```
// Material material = GetMaterial();
string newMaterialName = "my new material2";
if (!document.Settings.Materials.Contains(newMaterialName))
{
   material = material.Duplicate(newMaterialName);
   Debug.Assert(newMaterialName == material.Name);
   document.Settings.Materials.Remove(newMaterialName);
   Debug.Assert(!document.Settings.Materials.Contains(newMaterialName));
}
```

Document.Delete is a more generic method. See the Editing Elements chapter for details.

**Note:** Though you can delete material using Document.Delete, the Materials collection is not updated immediately after it is called. As a result, it is easy to cause Material management inconsistencies in your Add-in application. The best solution is to only use Materials.Remove.

## 15.3 Element Material

One element can have several elements and components. For example, FamilyInstance has SubComponents and Wall has CompoundStructure which contain several CompoundStructureLayers. (For more details about SubComponents refer to the Component Elements chapter and refer to the Host Elements and Opening chapter for more information about CompoundStructure.)

In the Revit API, get an element's materials using the following guidelines:

- If the element contains elements, get the materials separately.
- If the element contains components, get the material for each component from the parameters or in specific way (see Material section in the Host Elements and Opening chapter).
- If the component's material returns null, get the material from the corresponding Element.Category sub Category.

### 15.3.1 Material in a Parameter

If the Element object has a Parameter where ParameterType is ParameterType.Material, get the element material from Parameter. For example, a structural column FamilySymbol (a FamilyInstance whose Category is BuiltInCategory.OST_StructuralColumns) has the Column Material parameter. Get the Material using the ElementId. The following code example illustrates how to get structural column Material that has one component.

**Code Region 10- Material**

```
// The element should be a structural column.
ParameterSet parameterSet = element.Parameters;
string materialParaName = "Column Material";
foreach (Parameter parameter in parameterSet)
{
  if (parameter.Definition.Name == materialParaName)
  {
     Debug.Assert(parameter.Definition.ParameterType == ParameterType.Material);
    ElementId materialId = parameter.AsElementId();
    if (-1 == materialId.Value)
    {
      //Invalid ElementId, assume the material is "By Category"
      if (null != element.Category)
      {
        Material material = element.Category.Material;
      }
    }
    else
    {
       Autodesk.Revit.Elements.Material material =
         document.Settings.Materials.get_Item(materialId);
    }
```

```
      break;
   }
}
```

**Note:** If the material property is set to By Category in the UI, the ElementId for the material is -1 and cannot be used to retrieve the Material object as shown in the sample code. Try retrieving the Material from Category as described in the next section.

Some material properties contained in other compound parameters are not accessible from the API. As an example, in the following picture, for System Family: Railing, the Rail Structure parameter's StorageType is StorageType.None. As a result, you cannot get material information in this situation.

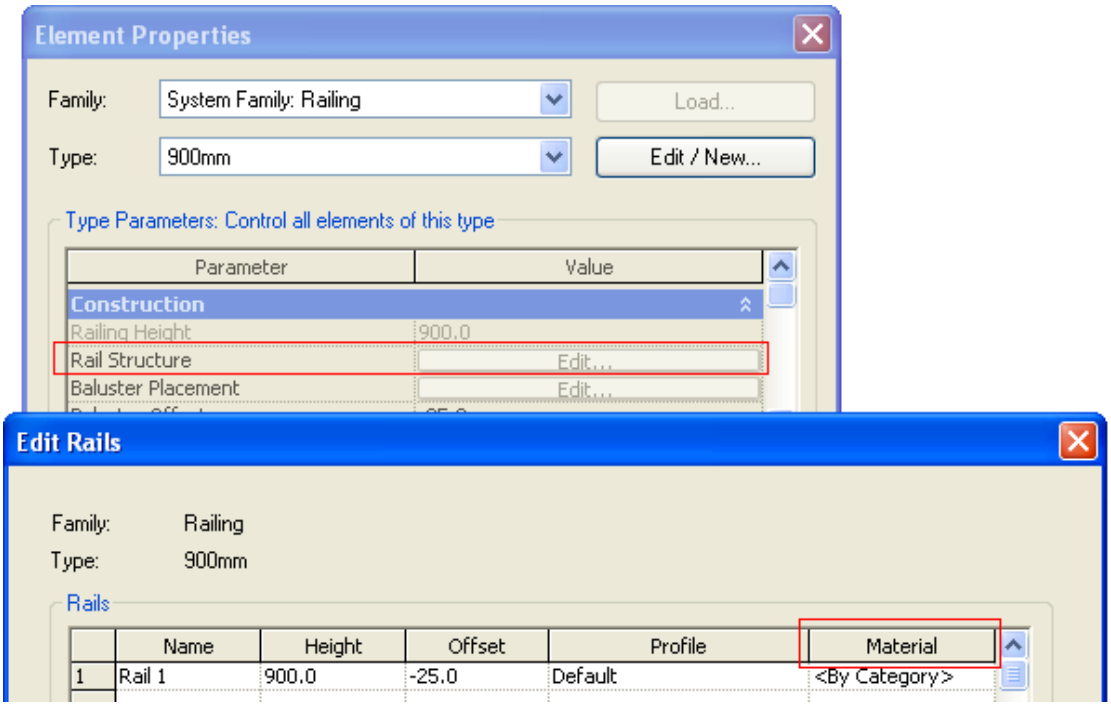



**Figure 148: Rail structure property**

### 15.3.2 Material and Category

Only model elements can have material.

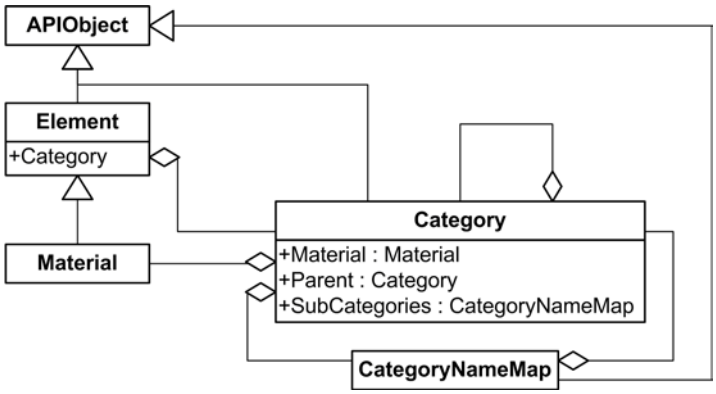



**Figure 149: Material and Category diagram**

From the Revit menu bar, click Settings > Object Style to display the Object Styles dialog box. Elements whose category is listed in the Model Objects tab have material information.

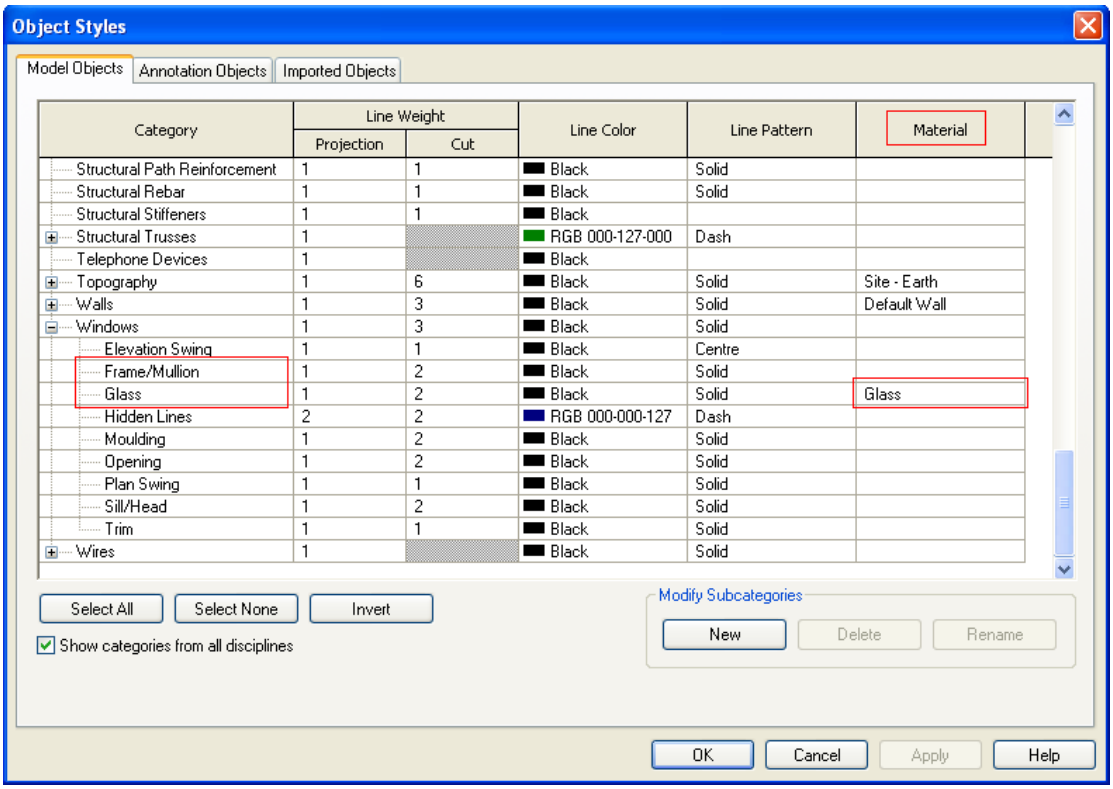



**Figure 150: Category material**

Only Model elements can have the Material property assigned. Querying Material for a category that corresponds to other than Model elements (e.g. Annotations or Imported) will therefore always result in a null (or Nothing in VB). For more details about the Element and Category classifications, refer to the Elements Essentials chapter.

If an element has more than one component, some of the Category.Subcategories correspond to the components.

In the previous Object Styles dialog box, the Windows Category and the Frame/Mullion and Glass subcategories are mapped to components in the windows element. In the following picture, it seems the window symbol Glass Pane Material parameter is the only way to get the window pane

material. However, the value is By Category and the corresponding Parameter returns -1 as an invalid ElementId.

In this case, the pane's Material is not null and it depends on the Category OST_WindowsFrameMullionProjection's Material property which is a subcategory of the window's category, OST_Windows. If it returns null as well, the pane's Material is determined by the parent cateogry OST_Windows. For more details, refer to the Walkthrough: Get Materials of a Window section in this chapter.

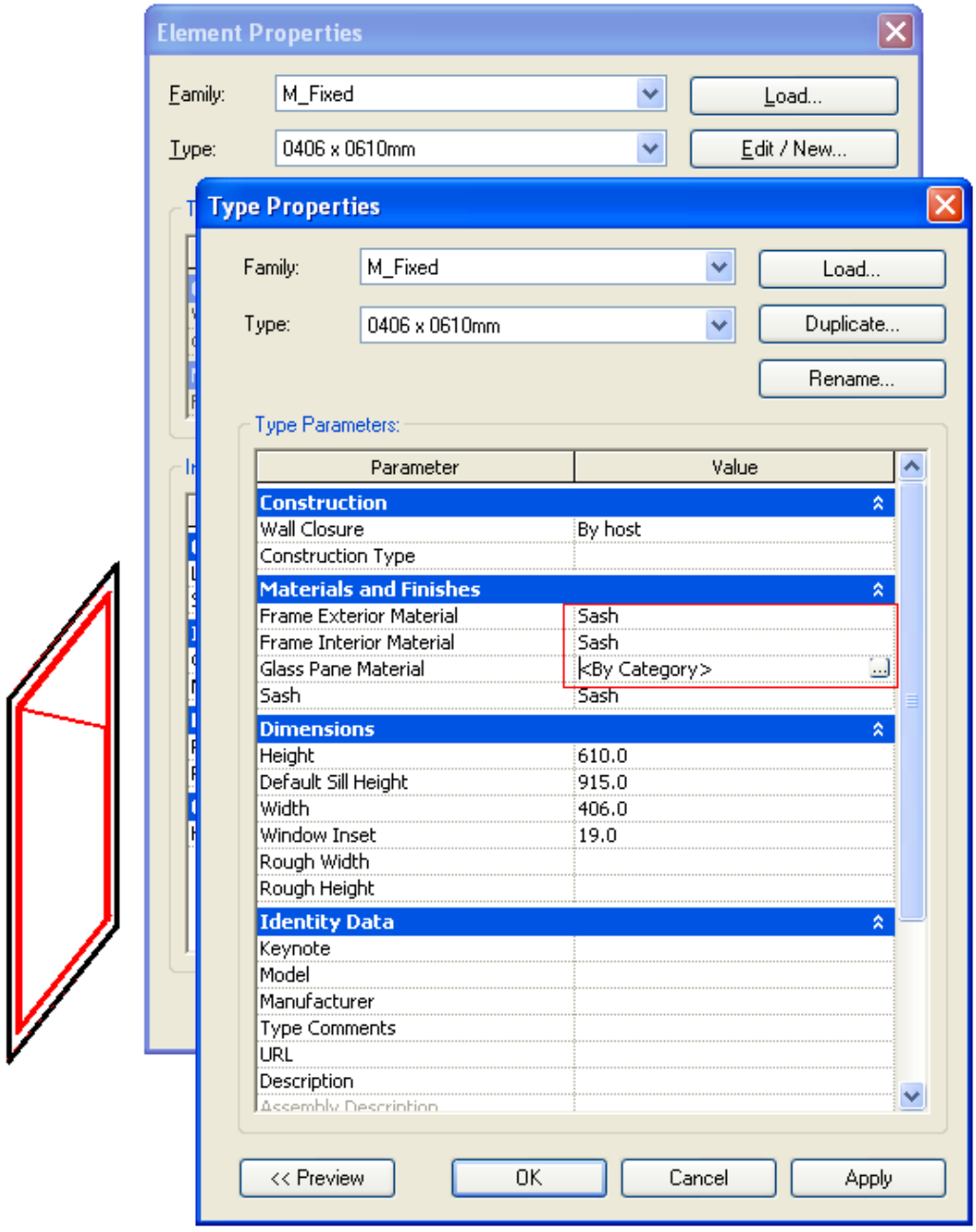



**Figure 151: Window material**

### 15.3.3 CompoundStructureLayer Material

You can get the CompoundStructureLayer object from HostObjAttributes. For more details, refer to the Host Elements and Opening chapter.

### 15.3.4 Retrieve Element Materials

The following diagram shows the workflow to retrieve Element Materials:

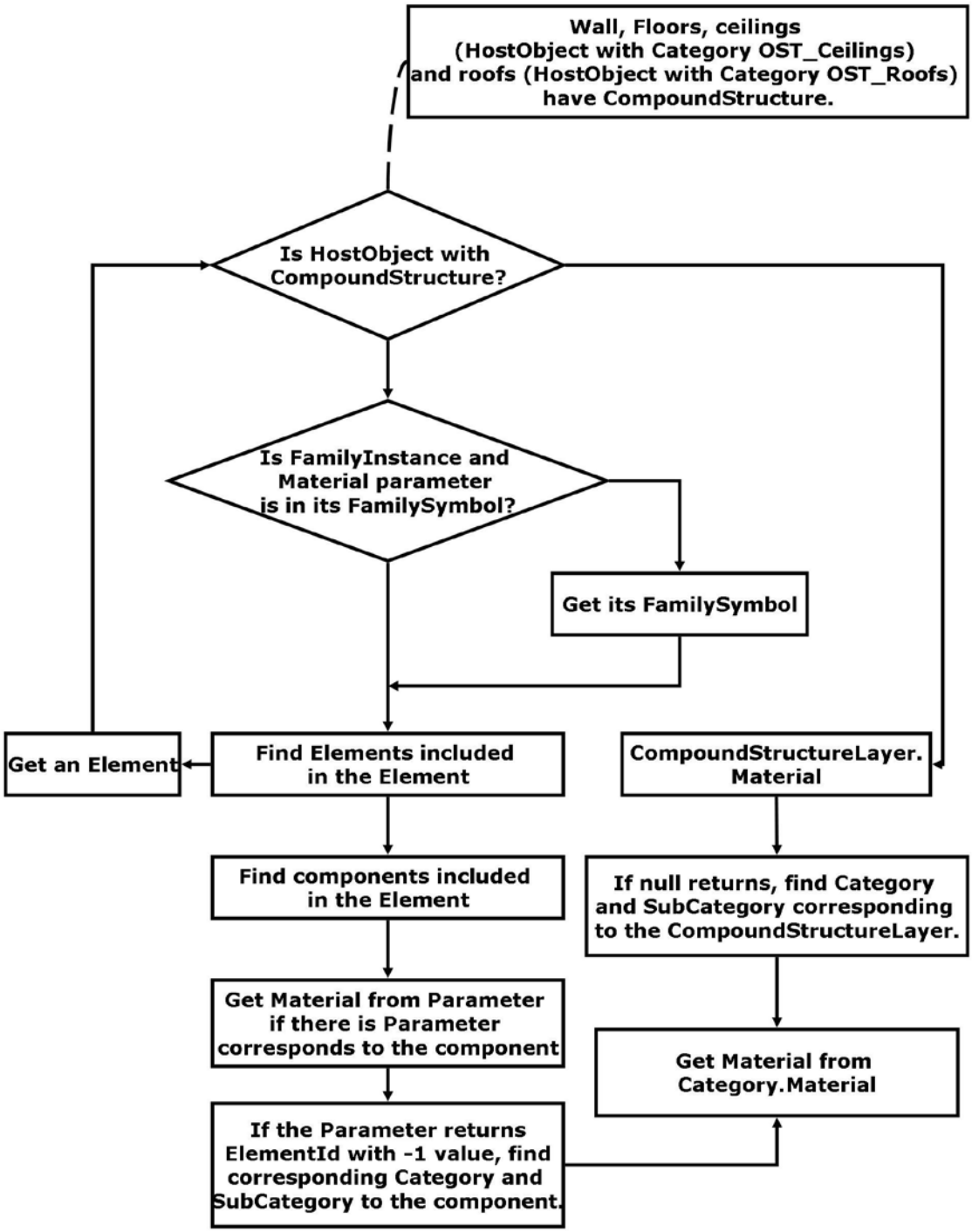

Wall, Floors, ceilings
(HostObject with Category OST_Ceilings)
and roofs (HostObject with Category OST_Roofs)
have CompoundStructure.
Is HostObject with CompoundStructure?
Is FamilyInstance and Material parameter is in its FamilySymbol?
Get its FamilySymbol
Get an Element
Find Elements included in the Element
CompoundStructureLayer. Material
Find components included in the Element
If null returns, find Category and SubCategory corresponding to the CompoundStructureLayer.
Get Material from Parameter if there is Parameter corresponds to the component
Get Material from Category.Material
If the Parameter returns ElementId with -1 value, find corresponding Category and SubCategory to the component.

**Figure 152: Get Element Material**

Workflow illustrates the following process:

- The workflow shows how to get the Elements.Material object (not Structural.Enums.Material enumerated type) that belongs to the element.
- There are two element classifications when retrieving the Material:

  - HostObject with CompoundStructure – Get the Elements.Material object from the CompoundStructureLayer class Material property.
  - Others - Get the Material from the Parameters.

  When you get a null Material object or an invalid ElementId with a value of -1, try the Material from the corresponding category. Note that a FamilyInstance and its FamilySymbol usually have the same category.

- The more you know about the Element object, the easier it is to get the material. For example:
  - If you know the Element is a beam, you can get the instance parameter Beam Material
  - If you know the element is a window, you can cast it to a FamilyInstance and get the FamilySymbol.

  After that you can get the Parameters such as Frame Exterior Material or Frame Interior Material to get the Elements.Material object. If you get null try to get the Material object from the FamilySymbol Category.
- Not all Element Materials are available in the API.

### 15.3.5 Walkthrough: Get Window Materials

The following code illustrates how to get the Window Materials.

**Code Region 11- Material**

```
// Select a window before run this command.
public class Command : Autodesk.Revit.IExternalCommand
{
public IExternalCommand.Result Execute(ExternalCommandData commandData, ref string message,
  ElementSet elements)
  {
    try
     {
        Autodesk.Revit.Document doc = commandData.Application.ActiveDocument;
        FamilyInstance window = null;
        foreach (Autodesk.Revit.Element element in doc.Selection.Elements)
        {
           window = element as FamilyInstance;
        }
       if (null == window)
            throw new Exception("Please select a window before run this commmand!");
       // Cast the window element to family instance
        Materials materials = doc.Settings.Materials;
        FamilySymbol windowSymbol = window.Symbol;
        Category category = windowSymbol.Category;
        Autodesk.Revit.Elements.Material frameExteriorMaterial = null;
        Autodesk.Revit.Elements.Material frameInteriorMaterial = null;
        Autodesk.Revit.Elements.Material sashMaterial = null;
        // Check the paramters firstly
        foreach (Parameter parameter in windowSymbol.Parameters)
        {
```

```
            switch (parameter.Definition.Name)
            {
               case "Frame Exterior Material":
                  frameExteriorMaterial = materials.get_Item(parameter.AsElementId());
                  break;
               case "Frame Interior Material":
                  frameInteriorMaterial = materials.get_Item(parameter.AsElementId());
                  break;
               case "Sash":
                sashMaterial = materials.get_Item(parameter.AsElementId());
                  break;
               default:
                  break;
            }
         }
         // Try category if the material is set by category
         if (null == frameExteriorMaterial)
            frameExteriorMaterial = category.Material;
         if (null == frameInteriorMaterial)
            frameInteriorMaterial = category.Material;
         if (null == sashMaterial)
            sashMaterial = category.Material;
         // Show the result
         // because the category may have a null Material, the got Material
         // objects need to be checked.
         string materialsInfo = "";
         materialsInfo += "Frame Exterior Material: " + (null != frameExteriorMaterial ?
         frameExteriorMaterial.Name : "null")   + "\n";
         materialsInfo += "Frame Interior Material: " + (null != frameInteriorMaterial ?
         frameInteriorMaterial.Name : "null")     + "\n";
         materialsInfo += "Sash: " + (null != sashMaterial ? sashMaterial.Name : "null") +
         "\n";
         MessageBox.Show(materialsInfo);
      }
      catch (Exception ex)
      {
         message = ex.Message;
         return IExternalCommand.Result.Failed;
      }
      return IExternalCommand.Result.Succeeded;
   }
}
```

# 16 Geometry

The Geometry namespace contains graphic-related types used to describe the graphical representation in the API. The API provides three classes used to describe and store the geometry information data according to their base classes:

- Geometry Node class – Includes classes derived from the GeometryObject class.
- Geometry Helper class – Includes classes derived from the APIObject class and value types
- Collection class – Includes classes derived from the IEnumerable or IEnumerator interface.

In this chapter, you learn how to use various graphic-related types, how to retrieve geometry data from an element, how to transform an element, and more.

## 16.1 Example: Retrieve Geometry Data from a Wall

This walkthrough illustrates how to get geometry data from a wall. The following information is covered:

- Getting the wall geometry edges.
- Getting the wall geometry faces.

**Note**: Retrieving the geometry data from Element in this example is limited because Instance is not considered. For example, sweeps included in the wall are not available in the sample code. The goal for this walkthrough is to give you a basic idea of how to retrieve geometry data but not cover all conditions. For more information about retrieving geometry data from Element, refer to Example: Retrieving Geometry Data from a Beam.

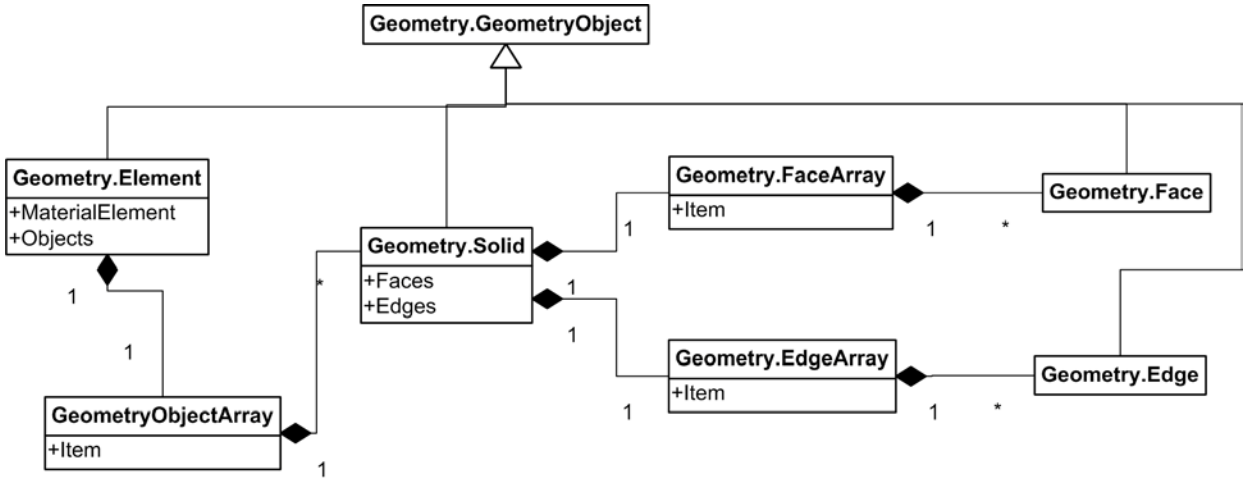



**Figure 153: Wall geometry diagram**

### 16.1.1 Create Geometry Options

In order to get the wall's geometry information, you must create a Geometry.Options object which provides detailed customized options. The detailed code is as follows:

**Code Region 1- Geometry**

```
Autodesk.Revit.Geometry.Options opt = commandData.Application.Create.NewGeometryOptions();
opt.ComputeReferences = true;
opt.DetailLevel = Autodesk.Revit.Geometry.Options.DetailLevels.Fine;
opt.View = commandData.Application.ActiveDocument.ActiveView;
```

**Note:** For more information, refer to the Geometry.Options section in this chapter.

### 16.1.2 Retrieve Faces and Edges

Wall geometry is a solid made up of faces and edges. Complete the following steps to get the faces and edges:

1. Retrieve a Geometry.Element instance using the Wall class Geometry property. This instance contains all geometry objects in the Object property, such as a solid, a line, and so on.
2. Iterate the Object property to get a geometry solid instance containing all geometry faces and edges in the Faces and Edges properties.
3. Iterate the Faces property to get all geometry faces.
4. Iterate the Edges property to get all geometry edges.

The sample code follows:

**Code Region 2- Geometry**

```
//Wall aWall = GetWall();
//Autodesk.Revit.Geometry.Options opt = GetGeometryOptions();
Autodesk.Revit.Geometry.Element geomElem = aWall.get_Geometry(opt);
foreach (GeometryObject geomObj in geomElem.Objects)
{
   Solid geomSolid = geomObj as Solid;
   if (null != geomSolid)
   {
      foreach (Face geomFace in geomSolid.Faces)
      {
         // get wall's geometry faces
      }
      foreach (Edge geomEdge in geomSolid.Edges)
      {
         // get wall's geometry edges
      }
   }
}
```

## 16.2 Geometry Node Class

The Geometry Node class describes the graphical representation in the API. Eight Geometry Node classes are in the API. The classes are:

- Profile - A geometric profile consisting of a loop of curves.
- Face - A 3D solid face. Some derived classes give detailed faces.
- Edge - A 3D solid edge.
- Curve - A parametric curve. Some derived classes provide detailed curves.
- Element - Geometric representation of an element containing all geometry information in its property. This element is not an Element type.
- Mesh - A triangular mesh used to describe the shape of a 3D face.

- Instance - An instance of another element (symbol). Specifically, you can retrieve a symbol's geometry information from the instance referring to it.
- Solid - 3D solid.

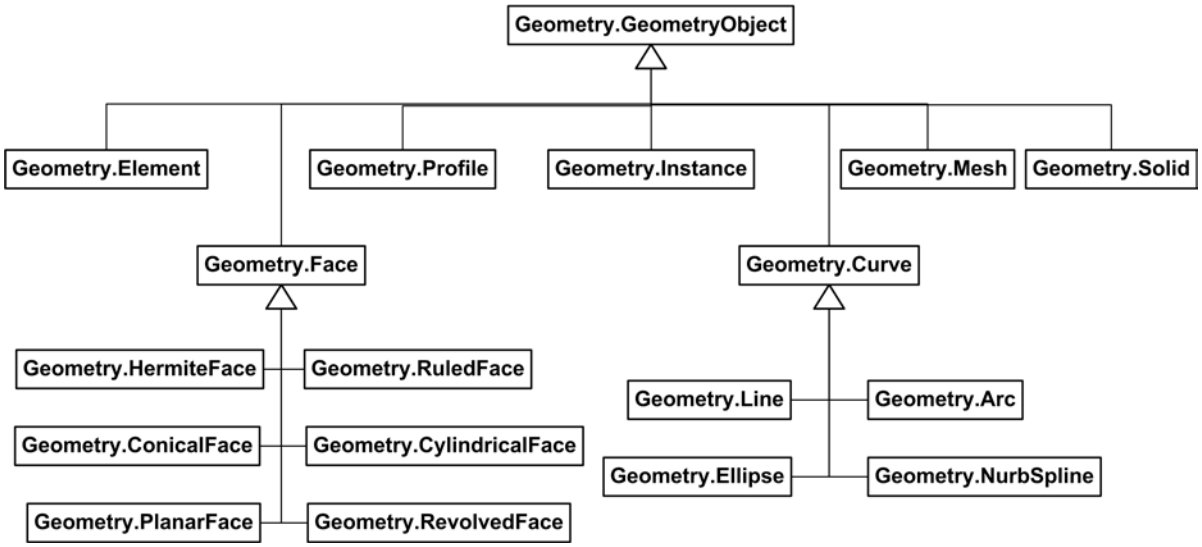



**Figure 154: Geometry node hierarchy diagram**

### 16.2.1 Geometry.Instance

The Instance class represents an instance of another element (symbol), specially positioned by this element. It is used to store geometry information for the symbol. The Instance class can pack any geometry information data, including another instance making it a good way to build a geometry tree using the instance in a complicated geometry representation.

The Instance class stores geometry data in the SymbolGeometry property using a local coordinate system. It also provides a Transform instance to convert the local coordinate system to a world coordinate space. To get the same geometry data as the Revit application from the Instance class, use the transform property to convert each geometry object retrieved from the SymbolGeometry property.

Generally, three geometry objects can be retrieved from an instance. They are:

- Curve – Curve or its derived class.
- Solid – Contains some faces and edges.
- Instance – Another instance which forms this instance.

Users need to transform the retrieved geometry objects using the Transfrom property. For more details, refer to the Geometry.Transform section in this chapter.

The following diagram illustrates how to get geometry objects from an instance.

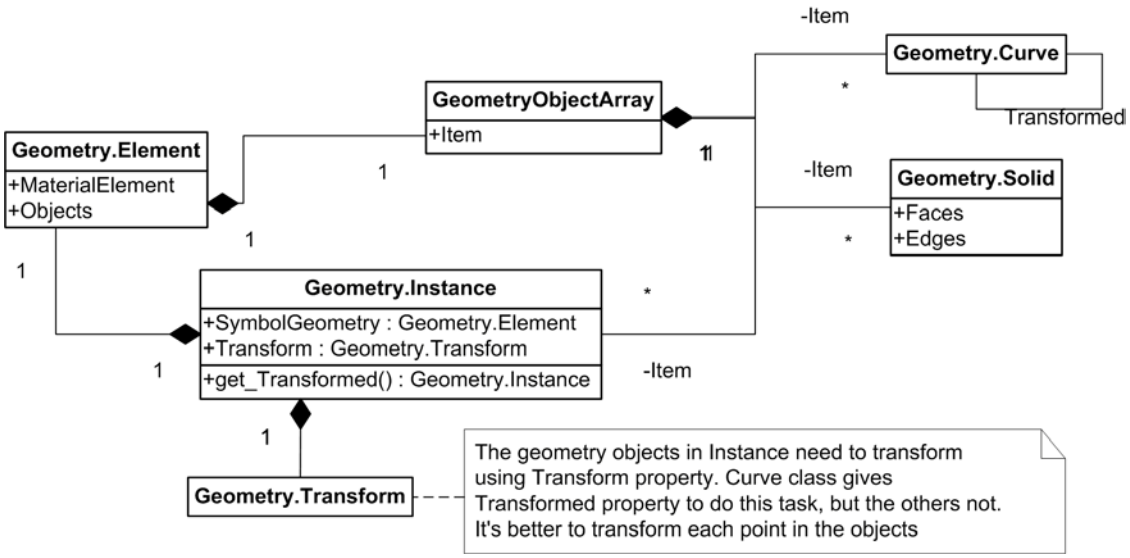



**Figure 155: Instance geometry diagram**

**Note:** The diagram identifies the main geometry objects you can retrieve in the Instance class. However, it does not cover all situations. Sometimes other geometry objects can be retrieved based on different elements in the Revit application.

Two samples are presented to explain the main usage.

#### 16.2.1.1 Sample 1

Get curves from an instance, and perform the coordinate transformation.

**Code Region 3- Geometry**

```
//Autodesk.Revit.Geometry.Instance instance = GetInstance();
foreach (GeometryObject instObj in instance.SymbolGeometry.Objects)
{
   Curve curve = instObj as Curve;
   if (null != curve)
   {
      // transfrom the curve to make it in the instance's coordinate space
      curve = curve.get_Transformed(instance.Transform);
      // the other works
   }
}
```

**Note**: It is uncommon to find curves in an instance. If you want to retrieve a curve in an instance, edit a family to add a line.

#### 16.2.1.2 Sample 2

Get the solid information from an instance, and perform the coordinate transformation.

**Code Region 4- Geometry**

```
//Autodesk.Revit.Geometry.Instance instance = GetInstance();
foreach (GeometryObject instObj in instance.SymbolGeometry.Objects)
{
   Solid solid = instObj as Solid;
```

```
   if (null == solid || 0 == solid.Faces.Size || 0 == solid.Edges.Size)
   {
      continue;
   }

   Transform instTransform = instance.Transform;
   // Get the faces and edges from solid, and transform the formed points
   foreach (Face face in solid.Faces)
   {
      Mesh mesh = face.Triangulate();
      foreach (XYZ ii in mesh.Vertices)
      {
         XYZ point = ii;
         XYZ transformedPoint = instTransform.OfPoint(ref point);
      }
   }
   foreach (Edge edge in solid.Edges)
   {
      foreach (XYZ ii in edge.Tessellate())
      {
         XYZ point = ii;
         XYZ transformedPoint = instTransform.OfPoint(ref point);
      }
   }
}
```

**Note:** You find similar situations when you get geometry information for doors, windows, and other elements. For more details about the retrieved geometry, refer to Example: Retrieving Geometry Data from a Beam in this chapter.

### 16.2.2 Geometry.Mesh

Geometry mesh is a triangular mesh used to describe the shape of a 3D face. The face object can be a curved surface which can be described by its endpoints just like a polygon. As a result, the mesh object provides a way to split the face into many small triangles where the endpoints locate the split face.

The following code sample illustrates how to draw the geometry of a mass. The mass geometry faces are described by the mesh.

**Code Region 5- Geometry**

```
// Face aFace = GetFace();
// a face has a mesh, all meshes are made of triangles
Mesh mesh = aFace.Triangulate();
for (int i = 0; i < mesh.NumTriangles; i++)
{
   MeshTriangle triangular = mesh.get_Triangle(i);
   // Define a XYZ array which stores the triangle's endpoints
```

```
   XYZArray points = new XYZArray();
   for (int n = 0; n < 3; n++)
   {
      XYZ point = triangular.get_Vertex(n);
      points.Append(ref point);
   }
   // invoke the method to draw the triangle
   // DrawTriangle(points);
}
```

The following pictures illustrate how a mass looks in the Revit application (left picture) and the same mass drawn with the API (right picture).

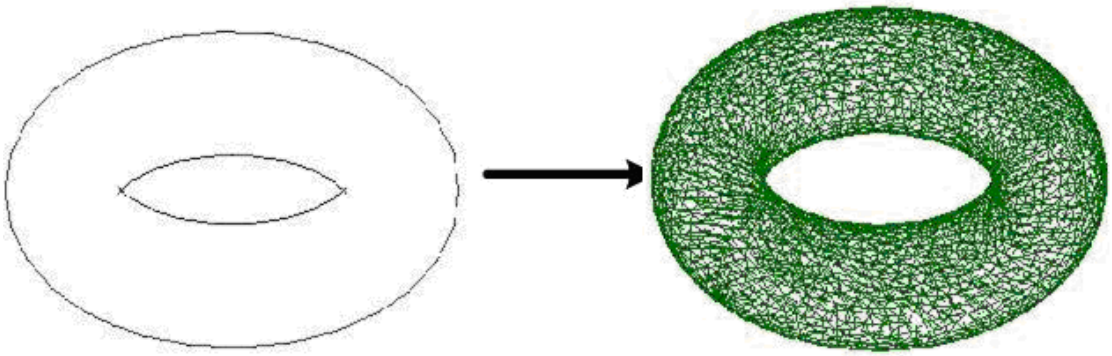


**Figure 156: Draw a mass using the API**

**Note**: The approximation tolerance used for display purposes is defined internally by Revit including distance tolerance and angle tolerance.

### 16.2.3 Solid

The Solid class defines a 3D solid from which users can get faces and edges. The Solid class defines a geometry solid, such as a cube or a cylinder. You can get the following using the solid's properties:

- Faces
- Edges
- Surface area
- Volume

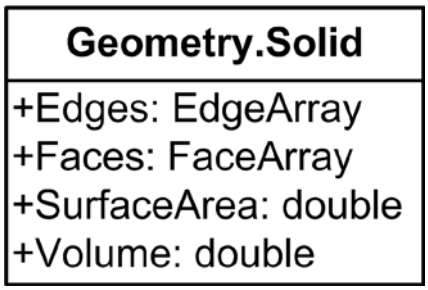



**Figure 157: Geometry.Solid class diagram**

**Note:** Sometimes the API can have unused solids containing zero edges and faces. Check the Edges and Faces properties before completing further work.

**Code Region 6- Geometry**

```
// GeometryObject geoObj = GetGeometryObject();
Solid solid = geoObj as Solid;
if(null != solid &&  0 != solid.Faces.Size)
{
   // do further work here.
}
```

## 16.3 Geometry Helper Class

Several Geometry Helper classes are in the API. The Help classes are used to describe geometry information for certain elements, such as defining a CropBox for a view using the BoundingBoxXYZ class.

- BoundingBoxXYZ - A 3D rectangular box used in many cases such as defining a 3D view section area.
- Transform - Transforming the affine 3D space.
- Reference - A stable reference to a geometric object in a Revit model, which is used when creating elements like dimensions.
- Plane – A completely flat surface in geometry.
- Options - User preferences for parsing geometry.
- XYZ Structure - Object representing coordinates in 3D space.
- UV Structure - Object representing coordinates in 2D space.
- BoundingBoxUV - A 2D rectangle parallel to the coordinate axes.

The following diagram displays the properties:

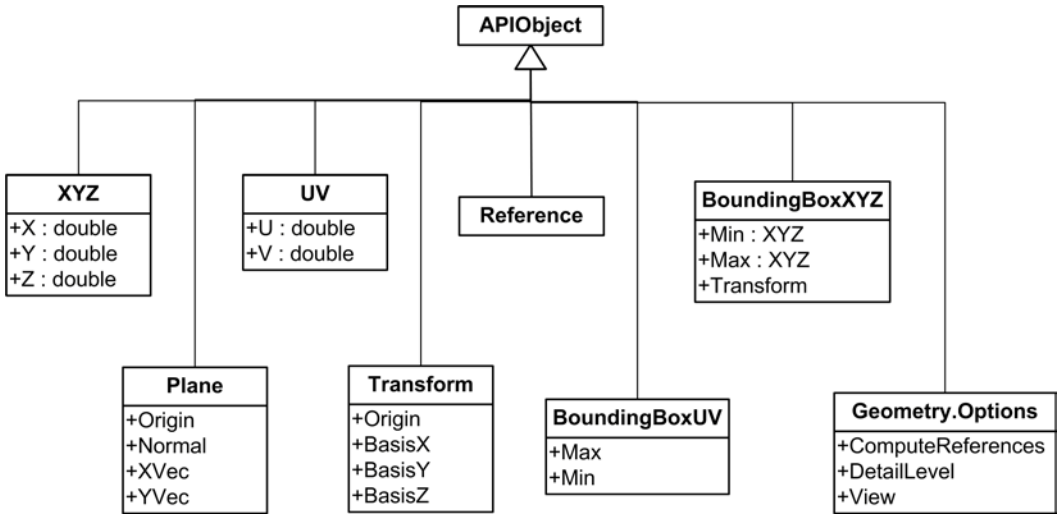



**Figure 158: Geometry helper hierarchy diagram**

### 16.3.1 Geometry.Transform

Transforms are limited to 3x4 transformations (Matrix) in the Revit application, transforming an object's place in the model space relative to the rest of the model space and other objects. The transforms are built from the position and orientation in the model space. Three direction Vectors

(BasisX, BasisY and BasisZ properties) and Origin point provide all of the transform information. The matrix formed by the four values is as follows:

$$\begin{pmatrix} \text{BasisX.X} & \text{BasisY.X} & \text{BasisZ.X} & \text{Origin.X} \\ \text{BasisX.Y} & \text{BasisY.Y} & \text{BasisZ.Y} & \text{Origin.Y} \\ \text{BasisX.Z} & \text{BasisY.Z} & \text{BasisZ.Z} & \text{Origin.Z} \end{pmatrix}$$

Applying the transformation to the point is as follows:

$$\begin{pmatrix} \text{BasisX.X} & \text{BasisY.X} & \text{BasisZ.X} & \text{Origin.X} \\ \text{BasisX.Y} & \text{BasisY.Y} & \text{BasisZ.Y} & \text{Origin.Y} \\ \text{BasisX.Z} & \text{BasisY.Z} & \text{BasisZ.Z} & \text{Origin.Z} \end{pmatrix} \times \begin{pmatrix} XYZ.X \\ XYZ.Y \\ XYZ.Z \\ 1 \end{pmatrix}$$

The Transform OfPoint method implements the previous function. The following code is a sample of the same transformation process.

**Code Region 7- Geometry**

```
public static XYZ TransformPoint(XYZ point, Transform transform)
{
   double x = point.X;
   double y = point.Y;
   double z = point.Z;

   //transform basis of the old coordinate system in the new coordinate // system
   XYZ b0 = transform.get_Basis(0);
   XYZ b1 = transform.get_Basis(1);
   XYZ b2 = transform.get_Basis(2);
   XYZ origin = transform.Origin;

   //transform the origin of the old coordinate system in the new
   //coordinate system
   double xTemp = x * b0.X + y * b1.X + z * b2.X + origin.X;
   double yTemp = x * b0.Y + y * b1.Y + z * b2.Y + origin.Y;
   double zTemp = x * b0.Z + y * b1.Z + z * b2.Z + origin.Z;

   return new XYZ(xTemp, yTemp, zTemp);
}
```

The Geometry.Transform class properties and methods are identified in the following sections.

#### 16.3.1.1 Identity

Transform the Identity.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

#### 16.3.1.2 Reflection

Reflect a specified plane.

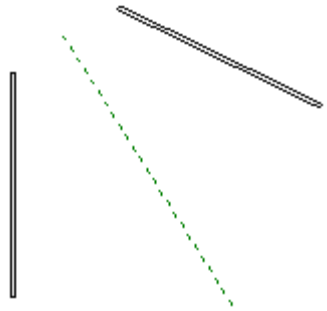


**Figure 159: Wall Reflection relationship**

As the previous picture shows, one wall is mirrored by a reference plane. The Reflection property needs the geometry plane information for the reference plane.

**Code Region 8- Geometry**

```
// ReferencePlane refPlane = GetRefPlane();
Transform mirTrans = Transform.get_Reflection(refPlane.Plane);
```

#### 16.3.1.3 Rotation

Rotate by a specified angle around a specified axis and point.

#### 16.3.1.4 Translation

Translate by a specified vector. Given a vector XYZ data, a transformation is created as follow:

$$\begin{pmatrix} x & y & z \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \end{pmatrix}$$

#### 16.3.1.5 Determinant

Transformation determinant.

$$\begin{vmatrix} \text{BasisX.X} & \text{BasisY.X} & \text{BasisZ.X} \\ \text{BasisX.Y} & \text{BasisY.Y} & \text{BasisZ.Y} \\ \text{BasisX.Z} & \text{BasisY.Z} & \text{BasisZ.Z} \end{vmatrix}$$

#### 16.3.1.6 HasReflection

Boolean value that indicates whether the transformation produces a reflection.

#### 16.3.1.7 Scale

Value that represents the transformation scale.

#### 16.3.1.8 Inverse

An inverse transformation. Transformation matrix A is invertible if a transformation matrix B exists such that A*B = B*A = I (identity).

#### 16.3.1.9 IsIdentity

Boolean value that indicates whether this transformation is an identity.

#### 16.3.1.10 IsTranslation

Boolean value that indicates whether this transformation is a translation.

Geometry.Transform provides methods to perform basal matrix operations.

#### 16.3.1.11 Multiply

Multiplies a transformation by a specified transformation and returns the result.

Operator* - Multiplies two specified transforms.

#### 16.3.1.12 ScaleBasis

Scales the basis vectors and returns the result.

#### 16.3.1.13 ScaleBasisAndOrigin

Scales the basis vectors and the transformation origin returns the result.

#### 16.3.1.14 OfPoint

Applies the transformation to the point. The Origin property is used.

#### 16.3.1.15 OfVector

Applies the transform to the vector. The Origin property is not used.

#### 16.3.1.16 AlmostEqual

Compares two transformations. AlmostEqual is consistent with the computation mechanism and accuracy in the Revit core code. Additionally, Equal and the == operator are not implemented in the Transform class.

The API provides several shortcuts to complete geometry transformation. The Transformed property in several geometry classes is used to do the work, as shown in the following table.

| Class Name | Function Description |
|---|---|
| Curve.get_Transformed(Transform transform) | Applies the specified transformation to a curve |
| Instance.get_Transformed(Transform transform) | Transforms the instance. |
| Profile.get_Transformed(Transform transform) | Transforms the profile and returns the result. |
| Mesh.get_Transformed(Transform transform) | Transforms the mesh and returns the result. |

**Table 40: Transformed Methods**

**Note:** The transformed method clones itself then returns the transformed cloned result.

### 16.3.2 Geometry.Reference

The Reference class does not contain properties or methods. However, it is very useful in element creation.

- Dimension creation requires references.
- The reference identifies a path within a geometric representation tree in a flexible manner.
- The tree is used to view specific geometric representation creation.

The API exposes four types of references based on different Pick pointer types. They are retrieved from the API in different ways:

1. For Point – Curve.EndPointReference property

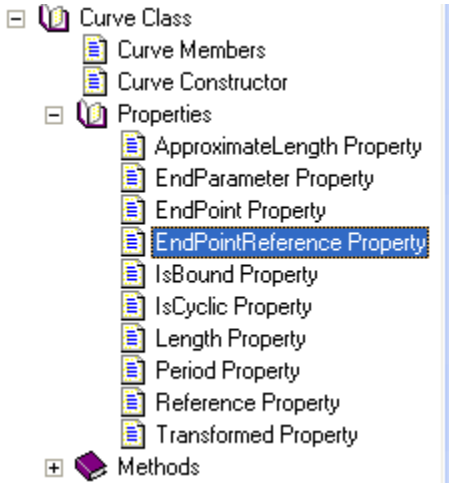



**Figure 160: Point reference**

2. For Curve (Line, Arc, and etc.) - Curve.Reference property

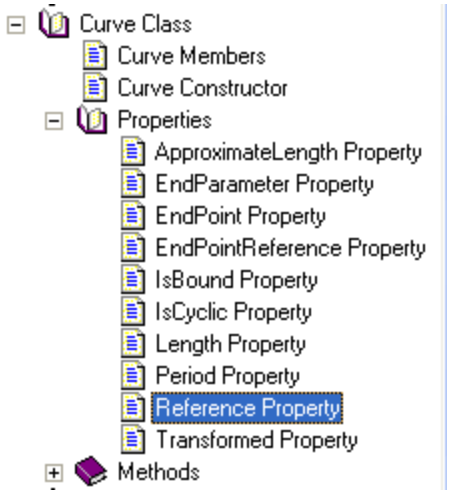



**Figure 161: Curve reference**

3. For Face – Face.Reference property

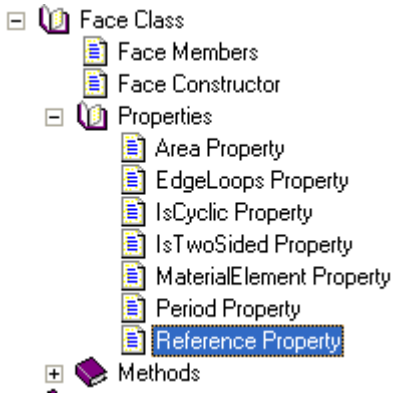



**Figure 162: Face reference**

4. For Cut Edge – Edge.Reference property

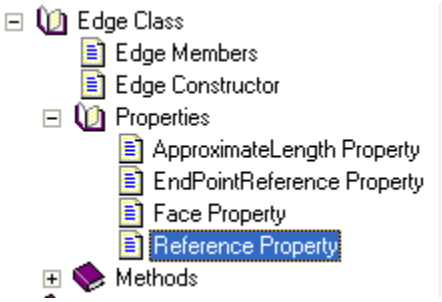



**Figure 163: Edge reference**

Different reference types cannot be used arbitrarily. For example:

- NewLineBoundaryConditions method requires a reference for Line
- NewAreaBoundaryConditions method requires a reference for Face
- NewPointBoundaryConditions method requires a reference for Point.

## 16.3.3 Geometry.Options

The Geometry.Options object defines a user preference for parsing geometry. It provides the following customized options:

- ComputeReferences – Indicates whether to compute the geometry reference when retrieving geometry information.
- View - Gets geometry information from a special view.
- DetailLevel – Indicates the preferred detail level.

### 16.3.3.1 ComputeReferences

If you set this property to false, the API does not compute a geometry reference. All Reference properties retrieved from the geometry tree return nothing. For more details about references, refer to the Reference section.

### 16.3.3.2 View

If users set the View property to a different view, the retrieved geometry information can be different. Review the following examples for more information:

1. In Revit, draw a stair in 3D view then select the Crop Region, Crop Region Visible, and Section Box properties in the 3D view. In the Crop Region, modify the section box in the 3D

view to display a portion of the stair. If you get the geometry information for the stair using the API and set the 3D view as the Options.View property, only a part of the stair geometry can be retrieved. The following pictures show the stair in the Revit application (left) and one drawn with the API (right).

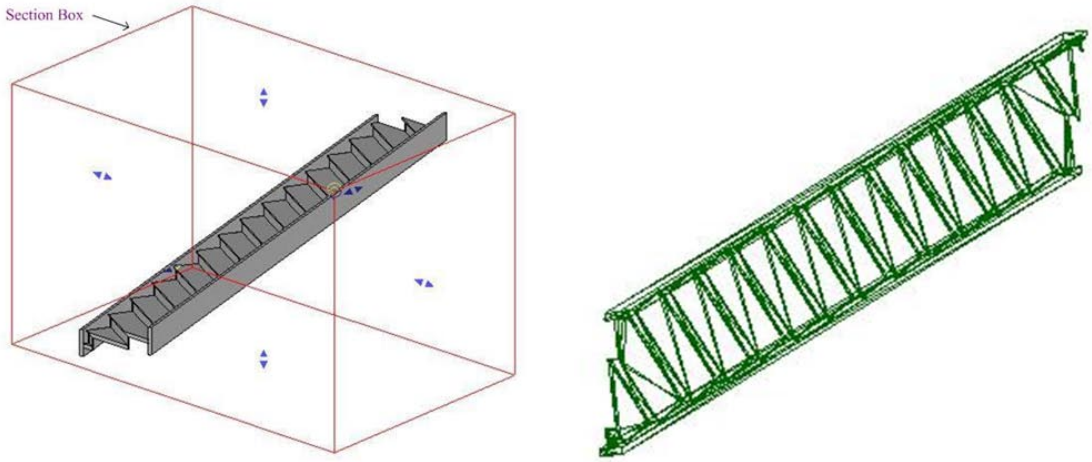



**Figure 164: Different section boxes display different geometry**

2. Draw a stair in Revit then draw a section as shown in the left picture. If you get the information for this stair using the API and set this section view as the Options.View property, only a part of the stair geometry can be retrieved. The stair drawn with the API is shown in the right picture.

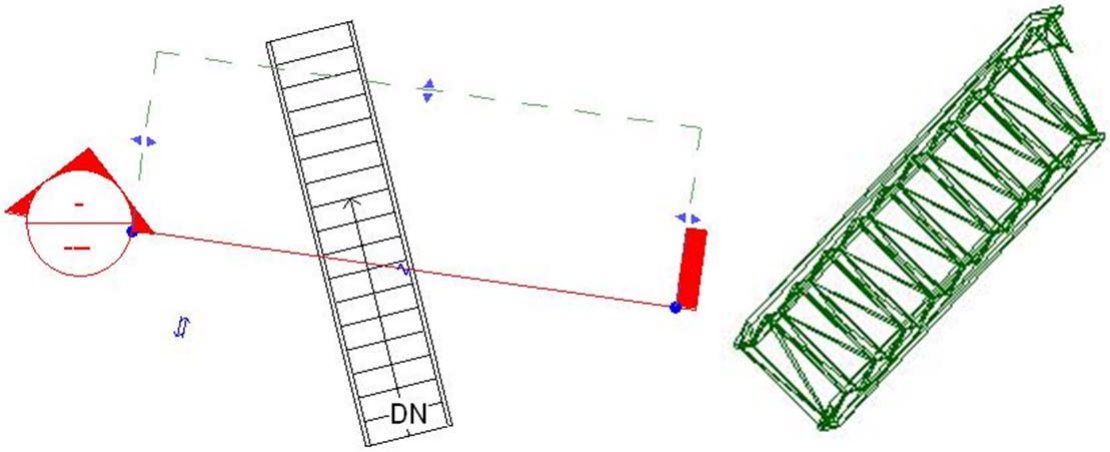



**Figure 165: Retrieve Geometry section view**

### 16.3.3.3 DetailLevel

The API defines three enumerations in Geometry.Options.DetailLevels. The three enumerations correspond to the three Detail Levels in the Revit application, shown as follows.

**Figure 166: Three detail levels**

Different geometry information is retrieved based on different settings in the DetailLevel property. For example, draw a beam in the Revit application then get the geometry from the beam using the API to draw it. The following pictures show the drawing results:

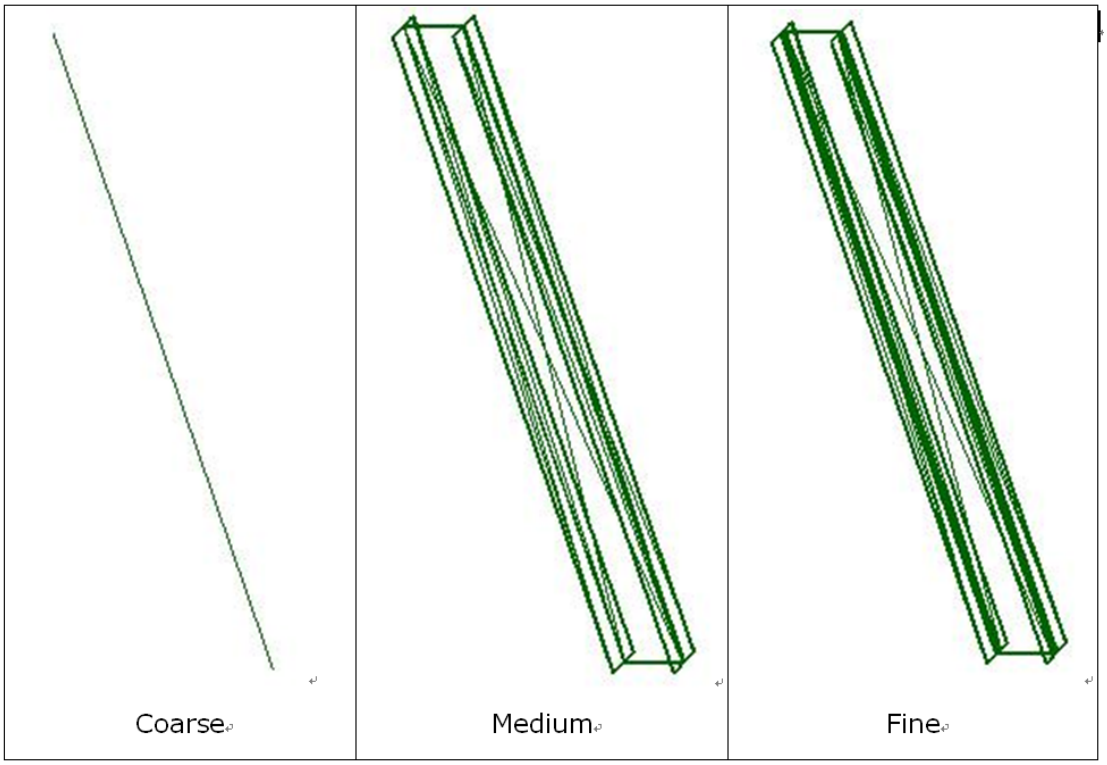



**Figure 167: Detail geometry for a beam**

#### 16.3.3.4 Creation

The Options object is created using the NewGeometryOptions method in the Creation.Application class. You can set properties after the instance is created.

**Code Region 9- Geometry**

```
Autodesk.Revit.Geometry.Options opt = commandData.Application.Create.NewGeometryOptions();
opt.ComputeReferences = true;
opt.DetailLevel = Autodesk.Revit.Geometry.Options.DetailLevels.Fine;
opt.View = commandData.Application.ActiveDocument.ActiveView;
```

### 16.3.4 Geometry.BoundingBoxXYZ

Different from BoundingBoxUV, BoundingBoxXYZ is a reference class derived from the APIObject class. It defines a 3D rectangular box that is required to be parallel to any coordinate axis. Similar to the Instance class, the BoundingBoxXYZ stores data in the local coordinate space. It has a Transform property that transforms the data from the box local coordinate space to the model space. In other words, to get the box boundary in the model space (the same one in Revit), transform each data member using the Transform property. The following sections illustrate how to use BoundingBoxXYZ.

#### 16.3.4.1 Define the View Boundaries

BoundingBoxXYZ can be used to define the view boundaries through View.CropBox property. The following pictures use a section view to show how BoundingBoxXYZ is used in the Revit application.

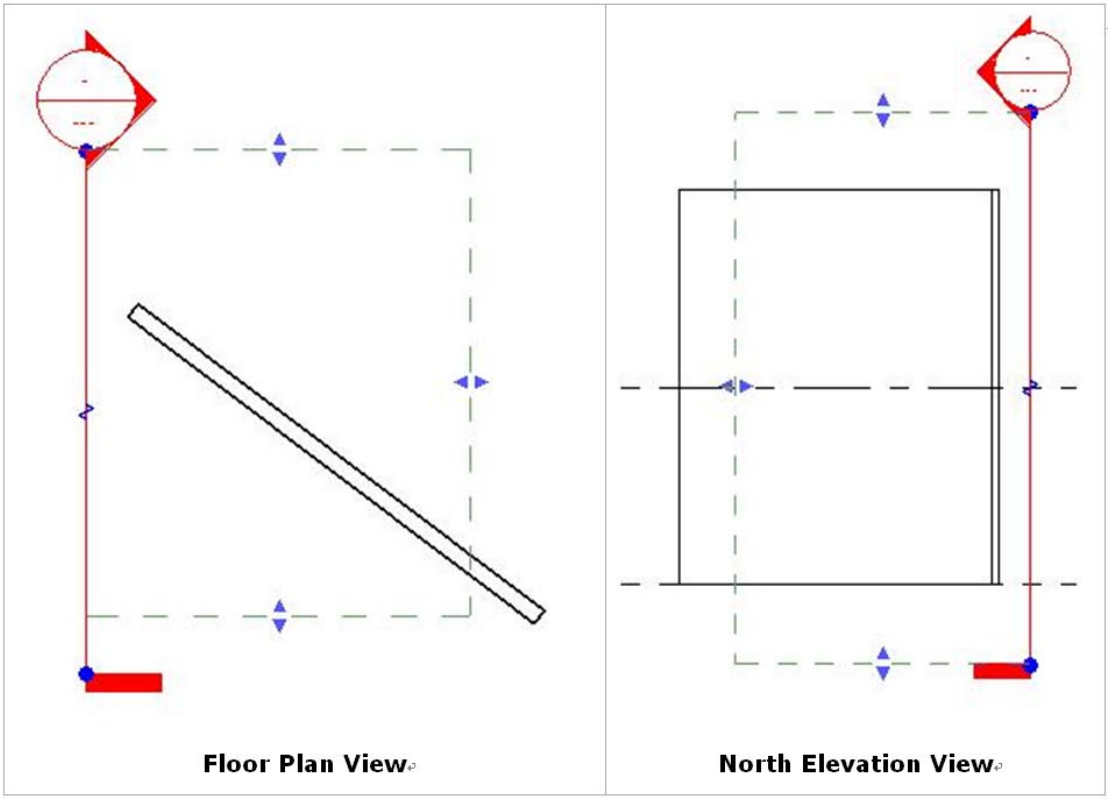



**Figure 168: BoundingBoxXYZ in section view**

The dash lines in the previous pictures show the section view boundary exposed as the CropBox property (a BoundingBoxXYZ instance).

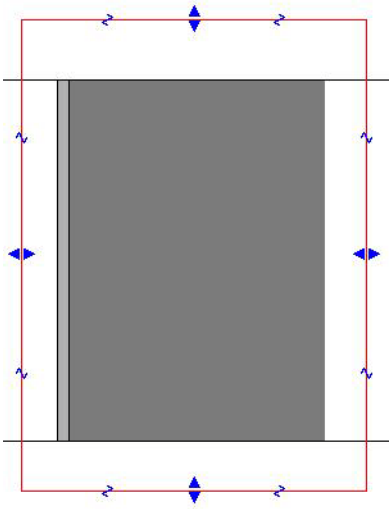


**Figure 169: Created section view**

The previous picture displays the corresponding section view. The wall outside the view boundary is not displayed.

### 16.3.4.2 Define a Section Box

BoundingBoxXYZ is also used to define a section box for a 3D view retrieved from the View3D.SectionBox property. Select the Section Box property in the Properties Dialog box. The section box is shown as follows:

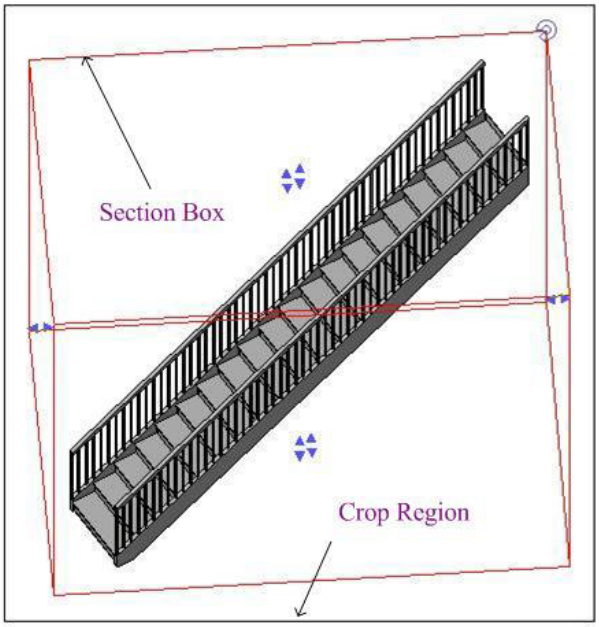



**Figure 170: 3D view section box**

#### 16.3.4.3 Other Uses

- Defines a box around an element's geometry. (Element.BoundingBox Property). The BoundingBoxXYZ instance retrieved in this way is parallel to the coordinate axes.
- Used in NewViewSection method in the Creation.Document class.

The following table identifies the main uses for this class.

| Property Name | Usage |
|---|---|
| Max/Min | Maximum/Minimum coordinates. Theses two properties define a 3D box parallel to any coordinate axis. The Transform property provides a transform matrix that can transform the box to the appropriate position. |
| Transform | Transform from the box coordinate space to the model space. |
| Enabled | Indicates whether the bounding box is turned on. |

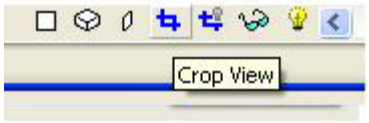
<table>
<tr><th>Property Name</th><th>Usage</th></tr>
<tr><td>MaxEnabled/ MinEnabled</td><td>Defines whether the maximum/minimum bound is active for a given dimension.
<ul>
<li>If the Enable property is false, these two properties should also return false.</li>
</ul>
<table>
<tr><td> 
</td><td>If the crop view is turned on, both MaxEnabled property and MinEnabled property return true.</td></tr>
<tr><td>
</td><td>If the crop view is turned off, both MaxEnabled property and MinEnabled property return false.</td></tr>
</table>
<ul>
<li>This property indicates whether the view's crop box face can be used to clip the element's view.</li>
<li>If BoundingBoxXYZ is retrieved from the View3D.SectionBox property, the return value depends on whether the Section Box property is selected in the 3D view Properties dialog box. If so, all Enabled properties return true.</li>
<li>If BoundingBoxXYZ is retrieved from the Element.BoundingBox property, all the Enabled properties are true.</li>
</ul></td></tr>
<tr><td>Bounds</td><td>Wrapper for the Max/Min properties.</td></tr>
<tr><td>BoundEnabled</td><td>Wrapper for the MaxEnabled/MinEnabled properties.</td></tr>
</table>

**Table 41: BoundingBoxXYZ properties**

The following code sample illustrates how to rotate BoundingBoxXYZ to modify the 3D view section box.

**Code Region 10- Geometry**

```
// View3D view3d = GetView3D();
BoundingBoxXYZ box = view3d.SectionBox;
if (false == box.Enabled)
{
    MessageBox.Show("The section box for View3D isn't Enable.");
    return;
}
// Create a rotation transform,
XYZ origin = new XYZ(0, 0, 0);
XYZ axis = new XYZ(0, 0, 1);
Transform rotate = Transform.get_Rotation(ref origin, ref axis, 2);
// Transform the View3D's SectionBox with the rotation transfrom
box.Transform = box.Transform.Multiply(rotate);
view3d.SectionBox = box;
```

### 16.3.5 Geometry.BoundingBoxUV

BoundingBoxUV is a value class that defines a 2D rectangle parallel to the coordinate axes. It supports the Min and Max data members. Together they define the BoundingBoxUV's boundary. BoundingBoxUV is retrieved from the View.Outline property which is the boundary view in the paper space view.

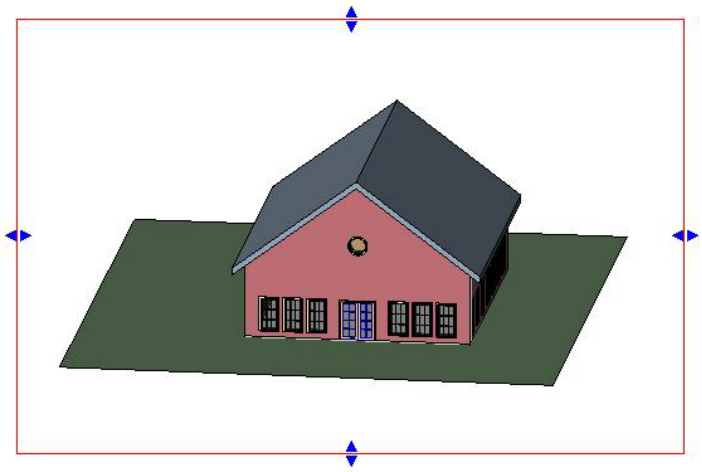


**Figure 171: View outline**

Two points define a BoundingBoxUV.

- Min point - The bottom-left endpoint.
- Max point - The upper-right endpoint.

**Figure 172: BoundingBoxUV Max and Min**

**Note**: BoundingBoxUV cannot present a gradient rectangle as the following picture shows.

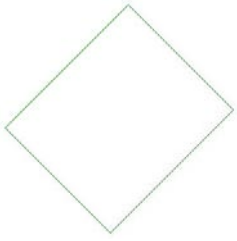


**Figure 173: Gradient rectangle**

## 16.4 Collection Classes

The API provides the following collection classes based on the items they contain:

| Class / Type | Corresponding Collection Classes | Corresponding Iterators |
|---|---|---|
| Curve | CurveArray | CurveArrayIterator |

| Class / Type | Corresponding Collection Classes | Corresponding Iterators |
|---|---|---|
| Edge | EdgeArray,<br>EdgeArrayArray | EdgeArrayIterator,<br>EdgeArrayArrayIterator |
| Face | FaceArray | FaceArrayIterator |
| GeometryObject | GeometryObjectArray | GeometryObjectArrayIterator |
| Instance | InstanceArray | InstanceArrayIterator |
| Mesh | MeshArray | MeshArrayIterator |
| Reference | ReferenceArray | ReferenceArrayIterator |
| Solid | SolidArray | SolidArrayIterator |
| UV | UVArray | UVArrayIterator |
| XYZ | XYZArray | XYZArrayIterator |
| Double value | DoubleArray | DoubleArrayIterator |

**Table 42: Collection Classes**

All of these classes use very similar methods and properties to do similar work. For more details, refer to the Collection chapter.

## 16.5 Example: Retrieve Geometry Data from a Beam

This section illustrates how to get solids and curves from a beam. You can retrieve column and brace geometry data in a similar way.

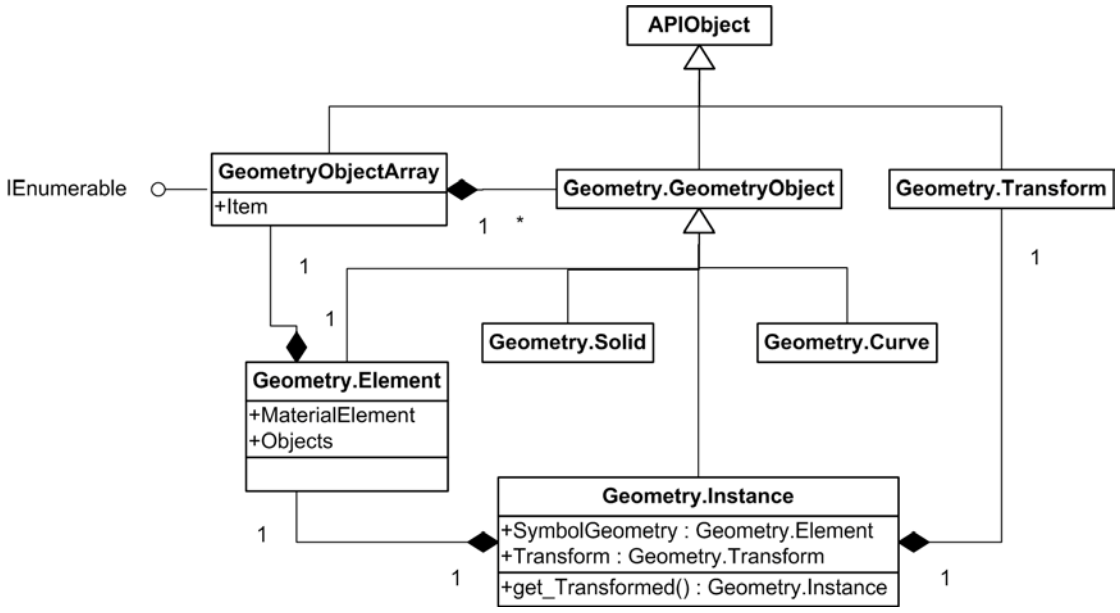



**Figure 174: Beam geometry diagram**

**Note:** If you want to get the beam and brace driving curve, call the FamilyInstance Location property where a LocationCurve is available.

The sample code is shown as follows:

**Code Region 11- Geometry**

```
// Get solids and curves from a beam


//[in]geomElem, [out]curves, [out]solids
```

```
public void GetCurvesFromABeam(
  Autodesk.Revit.Elements.FamilyInstance beam,
  Autodesk.Revit.Geometry.Options options,
  Autodesk.Revit.Geometry.CurveArray curves,
  Autodesk.Revit.Geometry.SolidArray solids)
{

   Autodesk.Revit.Geometry.Element geomElem
     = beam.get_Geometry(options);

   //Find all solids and insert them into solid array
   AddCurvesAndSolids(geomElem, curves, solids);
}

private void AddCurvesAndSolids(
  Autodesk.Revit.Geometry.Element geomElem,
  Autodesk.Revit.Geometry.CurveArray curves,
  Autodesk.Revit.Geometry.SolidArray solids)
{
   foreach (Autodesk.Revit.Geometry.GeometryObject geomObj in geomElem.Objects)
   {
      Autodesk.Revit.Geometry.Curve curve = geomObj as Autodesk.Revit.Geometry.Curve;
      if (null != curve)
      {
         curves.Append(curve);
         continue;
      }
      Autodesk.Revit.Geometry.Solid solid = geomObj as Autodesk.Revit.Geometry.Solid;
      if (null != solid)
      {
         solids.Append(solid);
         continue;
      }
      //If this GeometryObject is Instance, call AddCurvesAndSolids
      Autodesk.Revit.Geometry.Instance geomInst
        = geomObj as Autodesk.Revit.Geometry.Instance;
      if (null != geomInst)
      {
         Autodesk.Revit.Geometry.Instance transformedGeomInst
           = geomInst.get_Transformed(geomInst.Transform);
         AddCurves(transformedGeomInst.SymbolGeometry, curves, solids);
      }
   }
}
```

**Note**: For more information about how to retrieve the Geometry.Options type object, refer to the Geometry.Options section in this chapter.

# 17 Place and Locations

Every building has a unique place in the world because the Latitude and Longitude are unique. In addition, a building can have many locations in relation to other buildings. The Revit API Site namespace uses certain classes to save the geographical location information for Revit projects.

**Note:** Currently, the Revit API does not expose the Site menu functions. Only Site namespace provides functions corresponding to the menu options Setting > Manage Place and Locations.

The following diagram displays project location and associated classes in the Site namespace.

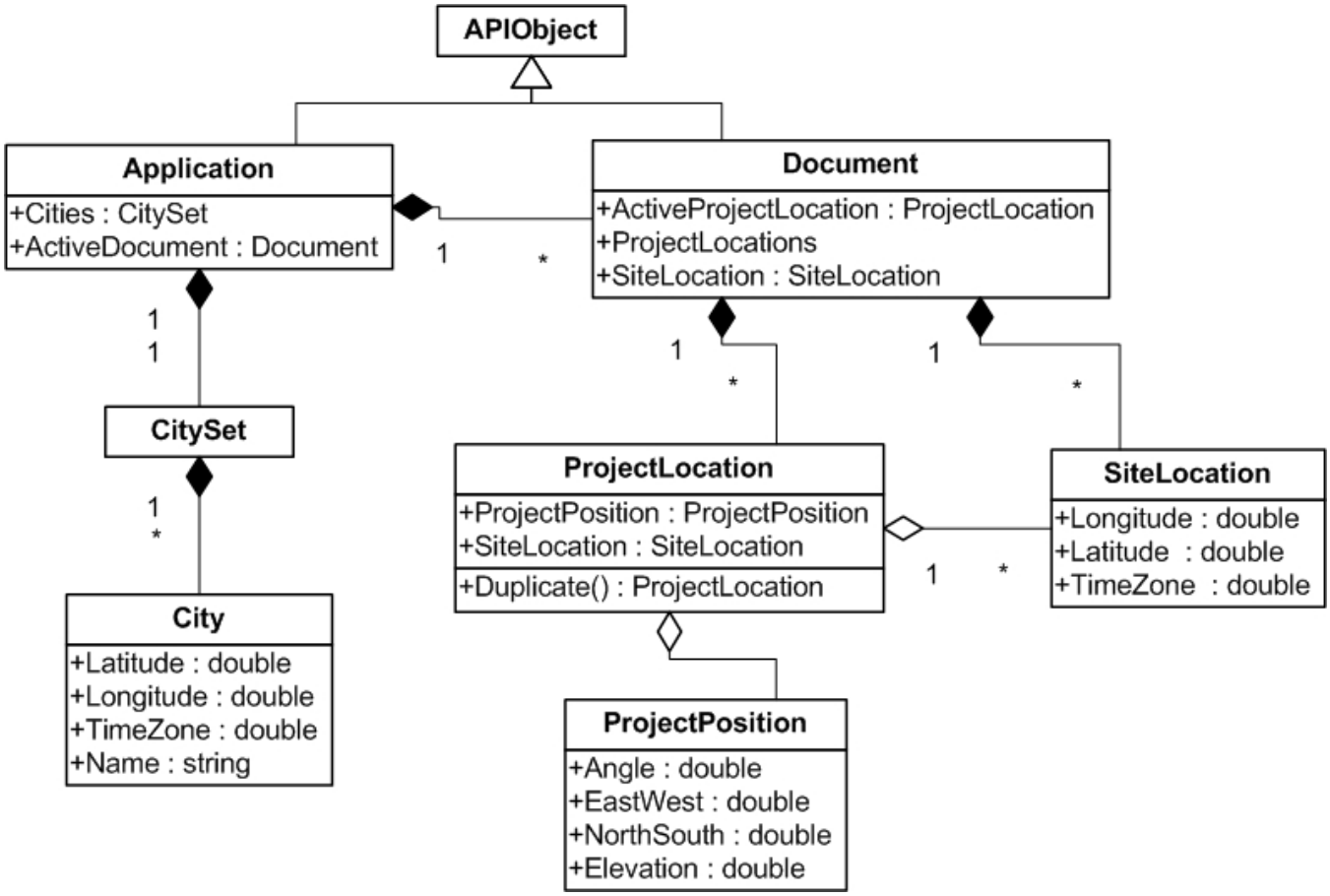



**Figure 175: Site-related class diagram**

## 17.1 Place

In the Revit API, the SiteLocation class contains place information including Latitude, Longitude, and Time Zone. This information identifies where the project is located in the world.

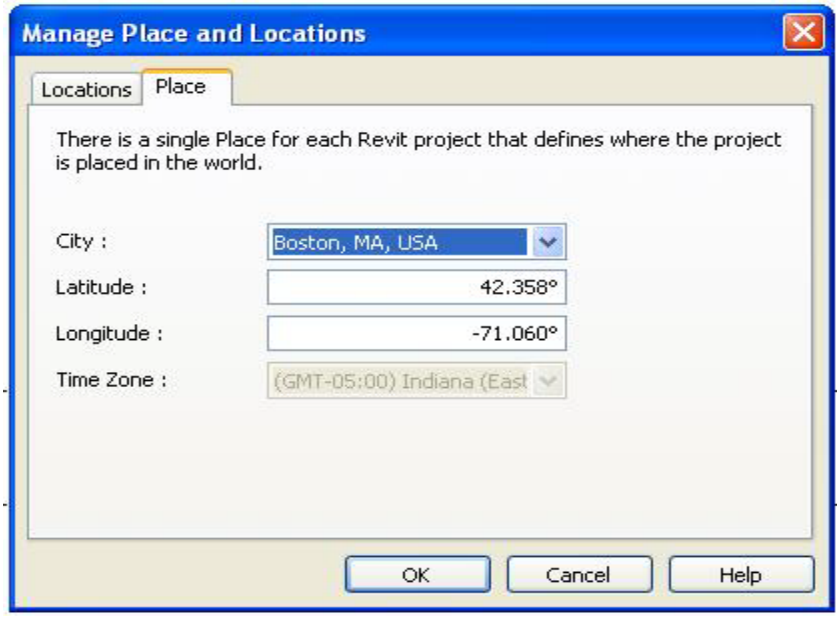



**Figure 176: Project Place**

Do not edit the Latitude, Longitude, and Time Zone fields even though the fields can be edited. If you change the Latitude or Longitude and the value is different from any city in the drop-down list, the City field is changed to User Defined.

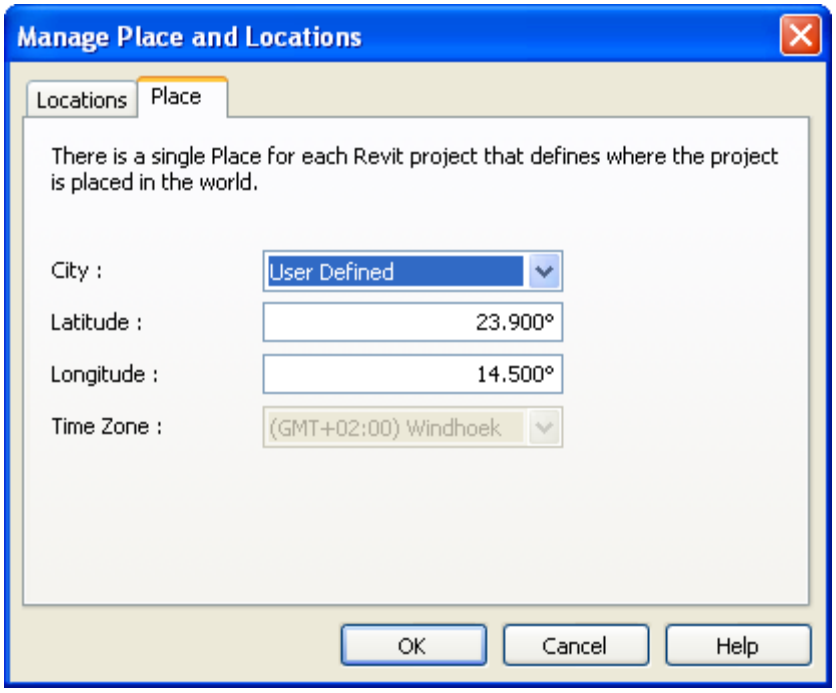



**Figure 177: Avoid changing the Latitude and Longitude**

**Note:** After you change the value of Latitude or Longitude, the City field is empty. Close the dialog box and open it again. The City field is changed to User Defined.

## 17.2 City

City is an object that contains geographical location information for a known city in the world. It contains longitude, latitude, and time zone information. The city list is retrieved by the Cities

property in the Application object. Currently, new cities cannot be added to the existing list in Revit. The city where the current project is located is not exposed by the Revit API.

## 17.3 ProjectLocation

A project only has one site which is the absolute location on the earth. However, it can have different locations relative to the projects around it. Depending on the coordinates and origins in use, there can be many ProjectLocation objects in one project.

By default each Revit project contains at least one named location, Internal. It is the active project location. You can retrieve it using the Document.ActiveProjectLocation property. All existing ProjectLocation objects are retrieved using the Document.ProjectLocations property.

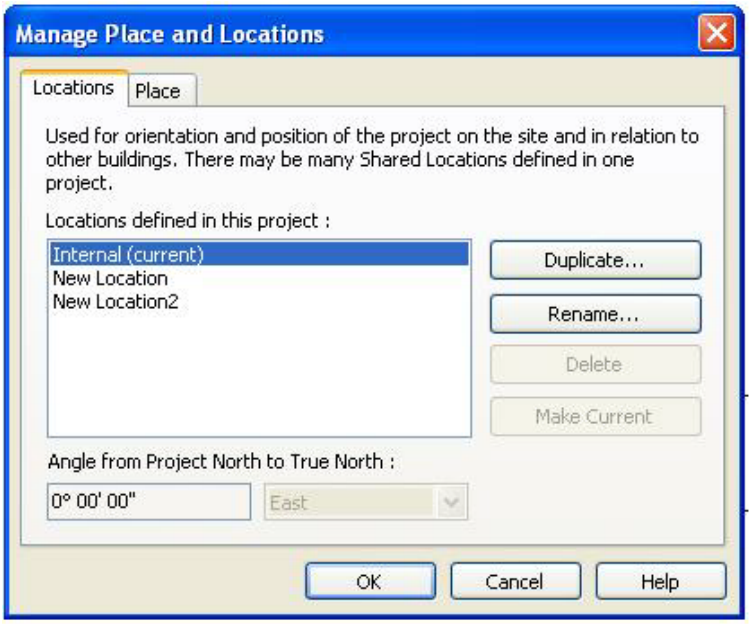



**Figure 178: Project locations**

## 17.4 Project Position

Project position is an object that represents a geographical offset and rotation. It is usually used by the ProjectLocation object to get and set geographical information. In the Revit UI, use the Tools > Shared Coordinates > Specify Shared Coordinates at a Point command to view the project location. The following pictures display the results after changing the ProjectLocation geographical rotation and the coordinates for the same point in the dialog box. However, you cannot see the result of changing the ProjectLocation geographical offset directly.

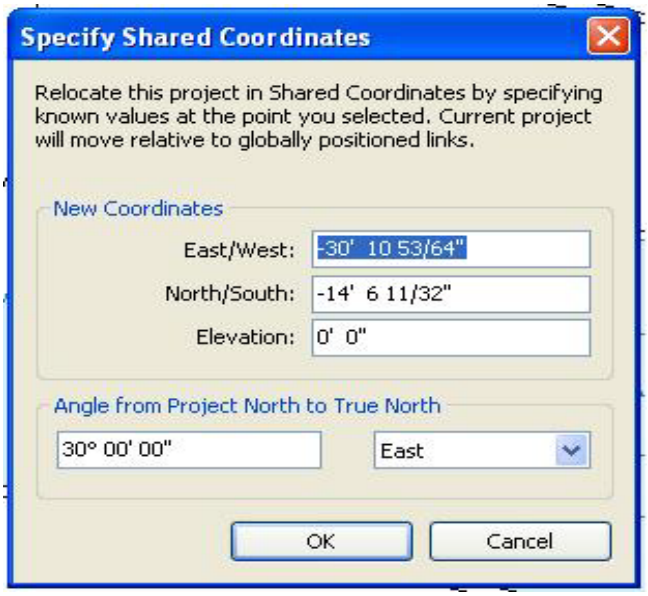



**Figure 179: Point coordinates**

**Note:** East indicates that the Location is rotated counterclockwise; West indicates that the location is rotated clockwise. If the Angle value is between 180 and 360 degrees, Revit transforms it automatically. For example, if you select East and type 200 degrees for Angle, Revit transforms it to West 160 degrees

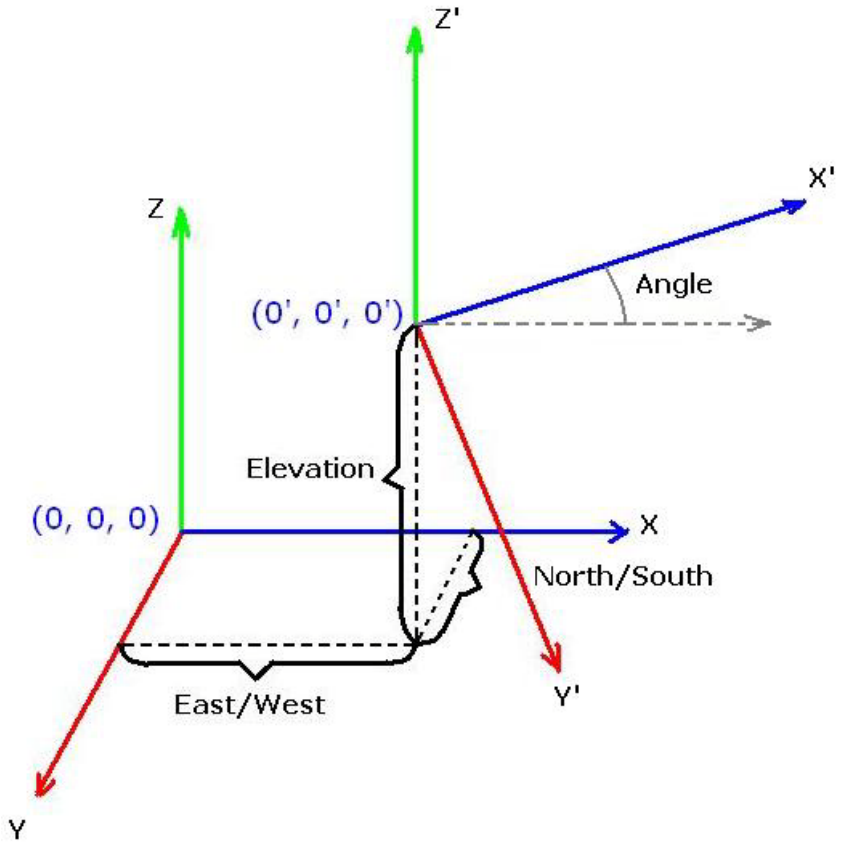



**Figure 180: Geographical offset and rotation sketch map**

The following sample code illustrates how to retrieve the ProjectLocation object and change the object's geographical offset and rotation.

**Code region 1 – Place and Locations**

```
//get and set the value of the projectlocation
public void GetProjectLocation(ExternalCommandData revit)
{
   ProjectLocation currentLocation = revit.Application.ActiveDocument.ActiveProjectLocation;
   //Retrieve all the project locations associated with this project
   ProjectLocationSet locations = revit.Application.ActiveDocument.ProjectLocations;

   //get the project position
   XYZ origin = new XYZ(0, 0, 0);
   //a modulus for degree convert to pi
   const double Modulus = 0.0174532925199433;
   ProjectPosition projectPosition =
     currentLocation.get_ProjectPosition(ref origin);
  //Angle from True North
   double angle = 30.0 * Modulus;
   double eastWest = 30.0;     //East to West offset
   double northSouth = 23.32;   //North to South offset
   double elevation = 22.14;    //Elevation above ground level

   //create a new project position
   ProjectPosition newPosition =
     revit.Application.Create.NewProjectPosition(eastWest, northSouth, elevation, angle);

   if (null != newPosition)
   {
      //set the value of the project position
      currentLocation.set_ProjectPosition(ref origin, newPosition);
   }
}
```

**Note:** There is only one active project location at a time. To see the result after changing the ProjectLocation geographical offset and rotation, change the Orientation property from Project North to True North in the plan view Properties dialog box.

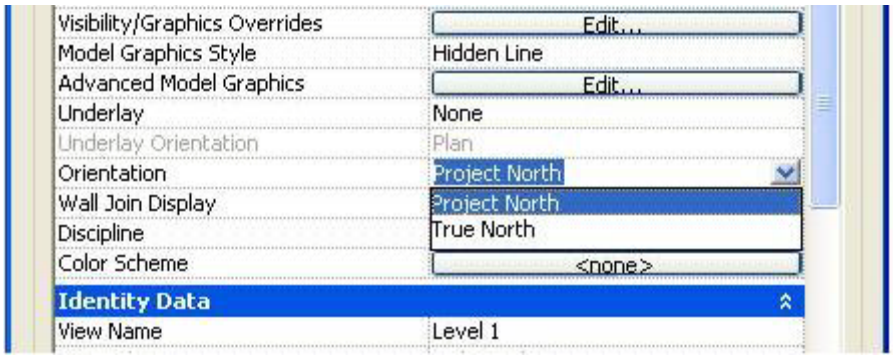



**Figure 181: Set orientation value in plan view Properties dialog box**

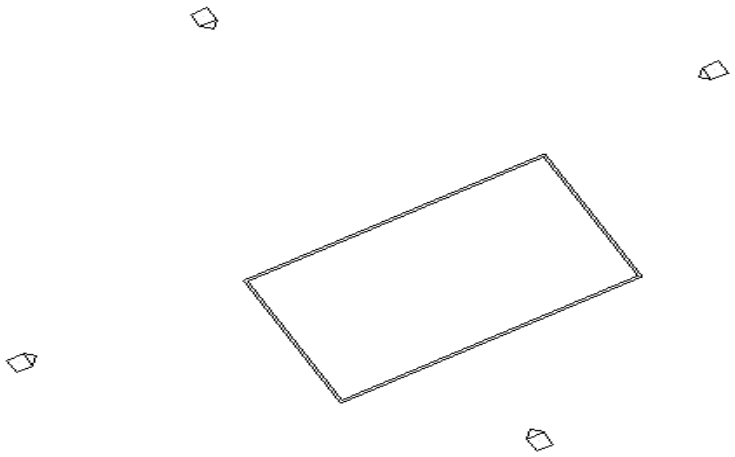


**Figure 182: Project is rotated 30 degrees from Project North to True North**

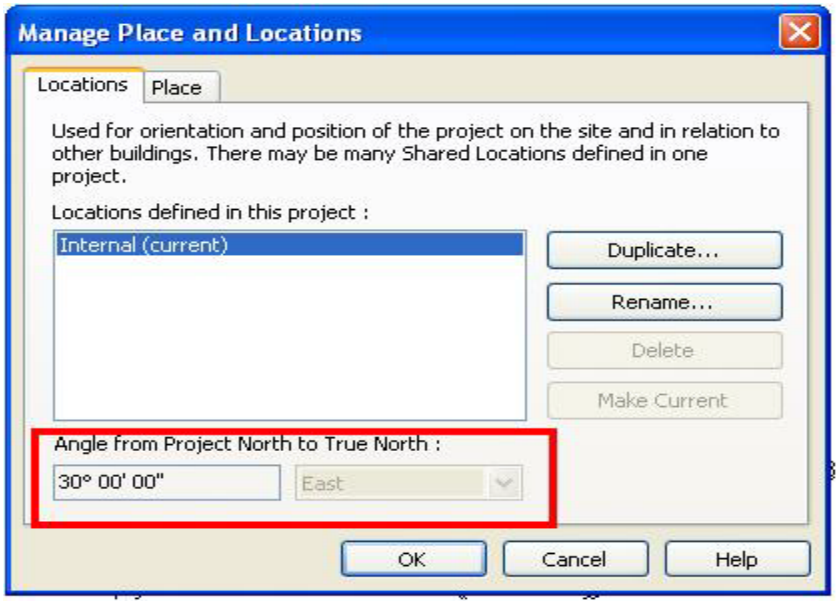



**Figure 183: Project location information**

### 17.4.1 Create and Delete

Create new project locations by duplicating an existing project location using the Duplicate method. The following code sample illustrates how to create a new project location using the Duplicate method.

**Code region 2 - Place and Locations**

```
public ProjectLocation DuplicateLocation(ExternalCommandData revit, string newName)
{
   ProjectLocation currentLocation = revit.Application.ActiveDocument.ActiveProjectLocation;
   ProjectLocationSet locations = revit.Application.ActiveDocument.ProjectLocations;
   foreach (ProjectLocation projectLocation in locations)
   {
      if (projectLocation.Name == newName)
```

```
      {
         throw new Exception("The name is same as a project location's name, please change
         one.");
      }
   }
   return currentLocation.Duplicate(newName);
}
```

The following code sample illustrates how to delete an existing project location from the current project.

**Code region 3 - Place and Locations**

```
public void DeleteLocation(ExternalCommandData revit)
{
   ProjectLocation currentLocation = revit.Application.ActiveDocument.ActiveProjectLocation;
   //There must be at least one project location in the project.
   ProjectLocationSet locations = revit.Application.ActiveDocument.ProjectLocations;
   if (1 == locations.Size)
   {
      return;
   }

   string name = "location";
   if (name != currentLocation.Name)
   {
      foreach (ProjectLocation projectLocation in locations)
      {
         if (projectLocation.Name == name)
         {
            revit.Application.ActiveDocument.Delete(projectLocation);
         }
      }
   }
}
```

**Note:** The following rules apply to deleting a project location:

- The active project location cannot be deleted because there must be at least one project location in the project.
- You cannot delete the project location because the ProjectLocationSet class instance is read-only.

# 18 Shared Parameter

Shared Parameters are parameter definitions stored in an external text file. The definitions are identified by a unique identifier generated when the definition is created and can be used in multiple projects.

This chapter introduces how to gain access to shared parameters through the Revit API. The following overview shows how to get a shared parameter and bind it to Elements in certain Categories:

- Set SharedParametersFileName
- Get the External Definition
- Binding

## 18.1 Definition File

The DefinitionFile object represents a shared parameter file. The definition file is a common text file. Do not edit the definition file directly; instead, edit it using the UI or the API.

### 18.1.1 Definition File Format

The shared parameter definition file is a text file (.txt) with two blocks: GROUP and PARAM.

**Definition File Sample**

```
# This is a Revit shared parameter file.
# Do not edit manually.
*GROUP ID     NAME
GROUP  1      MyGroup
GROUP  2      AnotherGroup
*PARAM GUID   NAME   DATATYPE      DATACATEGORY  GROUP  VISIBLE
PARAM  4b217a6d-87d0-4e64-9bbc-42b69d37dda6    MyParam       TEXT          1      1
PARAM  34b5cb95-a526-4406-806d-dae3e8c66fa9    Price  INTEGER             2      1
PARAM  05569bb2-9488-4be4-ae21-b065f93f7dd6    areaTags      FAMILYTYPE    -2005020      1
       1
```

- The GROUP block contains group entries that associate every parameter definition with a group. The following fields appear in the GROUP block:
  - ID - Uniquely identifies the group and associates the parameter definition with a group.
  - Name - The group name displayed in the UI.

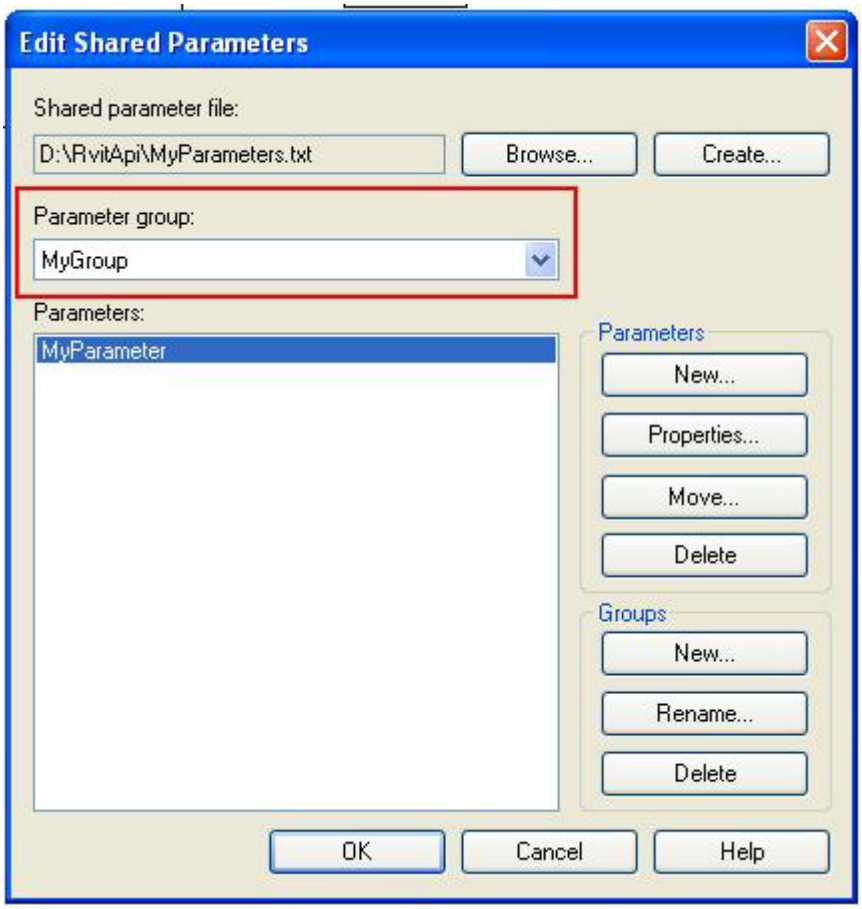



**Figure 184: Edit Shared Parameters (Group and Parameter)**

- The PARAM block contains parameter definitions. The following fields appear in the PARAM block:
  - GUID - Identifies the parameter definition.
  - NAME - Parameter definition name.
  - DATATYPE - Parameter type. This field can be a common type (TEXT, INTEGER, etc.), structural type (FORCE, MOMENT, etc.) or common family type (Area Tags, etc). Common type and structural type parameters are specified in the text file directly (e.g.: TEXT, FORCE). If the value of the DATATYPE field is FAMILYTYPE, an extra number is added. For example, FAMILYTYPE followed by -2005020 represents Family type: Area Tags.

| |
|---|
| FAMILYTYPE -2005020 |

**Figure 185: New parameter definition**

- GROUP - A group ID used to identify the group that includes the current parameter definition.
- VISIBLE - Identifies whether the parameter is visible. The value of this field is 0 or 1.

  0 = invisible

  1 = visible

As the definition file sample shows, there are two groups:

- MyGroup - ID 1 - Contains the parameter definition for MyParam which is a Text type parameter.
- AnotherGroup - ID 2 - Contains the parameter definition for Price which is an Integer type parameter.

## 18.2 Definition File Access

In the add-in code, complete the following steps to gain access to the definition file:

1. Specify the Autodesk.Revit.Options.Application.SharedParametersFilename property with an existing text file or a new one.

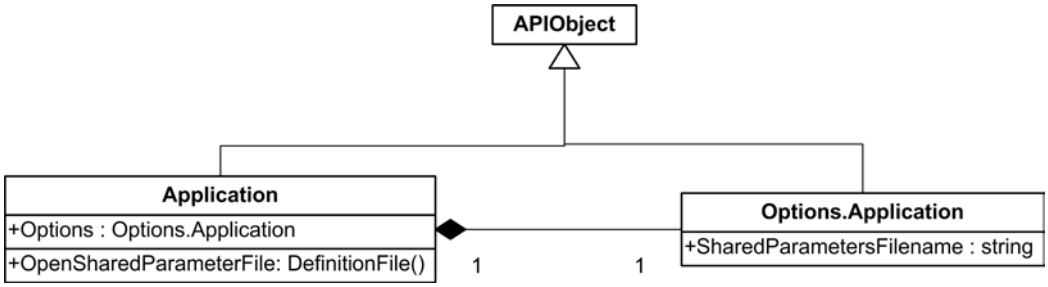



**Figure 186: Application and Options.Application diagram**

2. Open the shared parameters file, using the Application.OpenSharedParameterFile method.
3. Open an existing group or create a new group using the DefinitionFile.Groups property.
4. Open an existing external parameter definition or create a new definition using the DefinitionGroup.Definitions property.

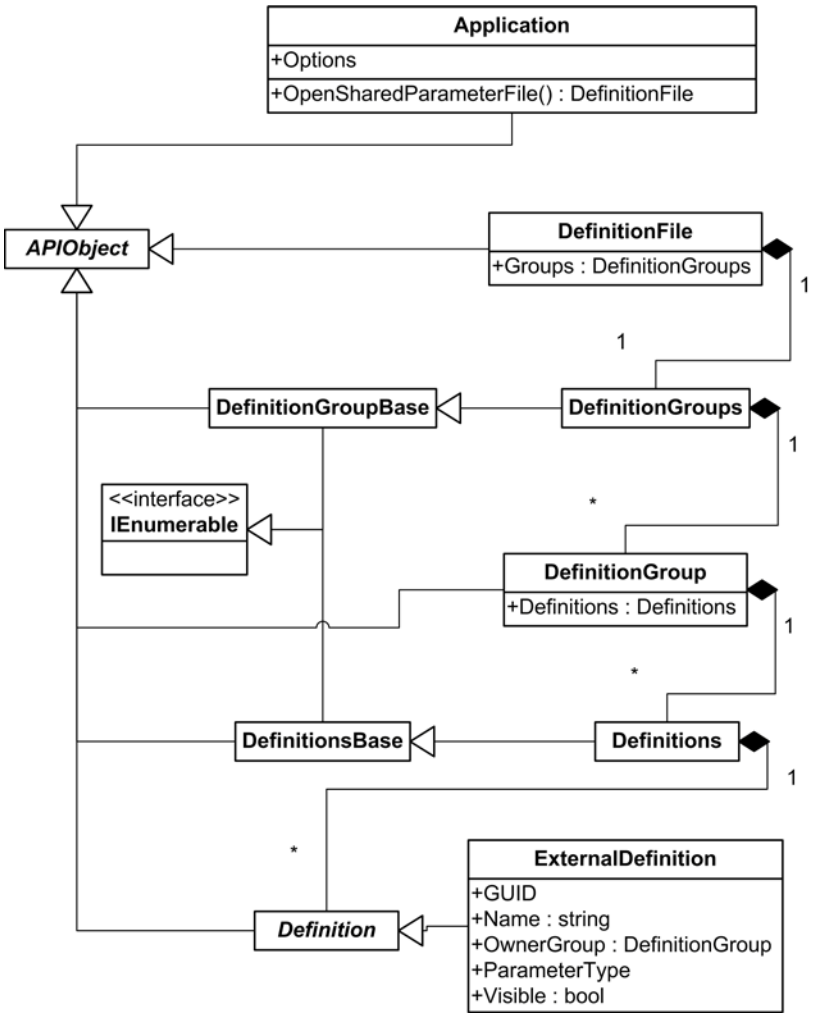



**Figure 187: Get the ExternalDefinition type object diagram**

The following list provides more information about the classes and methods in the previous diagram.

- Autodesk.Revit.Parameters.DefinitionFile Class - The DefinitionFile object represents one shared parameter file.
  - The object contains a number of Group objects.
  - Shared parameters are grouped for easy management and contain shared parameter definitions.
  - You can add new definitions as needed.
  - The DefinitionFile object is retrieved using the Application.OpenSharedParameterFile method.
- Autodesk.Revit.Parameters.ExternalDefinition Class - The ExternalDefinition class is derived from the Definition class.
  - The ExternalDefinition object is created by a DefinitionGroup object from a shared parameter file.
  - External parameter definitions must belong to a Group which is a collection of shared parameter definitions.

- Autodesk.Revit.Options.Application.SharedParametersFilename Property - Get and set the shared parameter file path using the Autodesk.Revit.Options.SharedParametersFilename property.
  - By default, Revit does not have a shared parameter file.
  - Initialize this property before using. If it is not initialized, an exception is thrown.
- Autodesk.Revit.Application.OpenSharedParameterFile Method - This method returns an object representing a Revit shared parameter file.
  - Revit uses one shared parameter file at a time.
  - The file name for the shared parameter file is set in the Revit Application Options object. If the file does not exist, an exception is thrown.

### 18.2.1 Create a Shared Parameter File

Because the shared parameter file is a text file, you can create it using code or create it manually.

**Code Region 1-SharedParameter**

```
private void CreateExternalSharedParamFile(string sharedParameterFile)
{
   System.IO.FileStream fileStream = System.IO.File.Create(sharedParameterFile);
   fileStream.Close();
}
```

### 18.2.2 Access an Existing Shared Parameter File

Because you can have many shared parameter files for Revit, it is necessary to specifically identify the file and external parameters you want to access. The following two procedures illustrate how to access an existing shared parameter file.

#### 18.2.2.1 Get DefinitionFile from an External Parameter File

Set the shared parameter file path to `app.Options.SharedParametersFilename` as the following code illustrates, then invoke the Autodesk.Revit.Application.OpenSharedParameterFile method.

**Code Region 2-SharedParameter**

```
private Autodesk.Revit.Parameters.DefinitionFile
  SetAndOpenExternalSharedParamFile (Autodesk.Revit.Application app, string
sharedParameterFile)
{
   // set the path of shared parameter file to current Revit
   app.Options.SharedParametersFilename = sharedParameterFile;
   // open the file
   return app.OpenSharedParameterFile();
}
```

**Note:** Consider the following points when you set the shared parameter path:

- During each installation, Revit cannot detect whether the shared parameter file was set in other versions. You must bind the shared parameter file for the new Revit installation again.
- If Options.SharedParametersFilename is set to a wrong path, an exception is thrown only when OpenSharedParameterFile is called.

- Revit can work with multiple shared parameter files. Even though only one parameter file is used when loading a parameter, the current file can be changed freely.

#### 18.2.2.2 Traverse All Parameter Entries

The following sample illustrates how to traverse the parameter entries and display the results in a message box.

**Code Region 3-SharedParameter**

```
private void ShowDefinitionFileInfo(
  Autodesk.Revit.Parameters.DefinitionFile myDefinitionFile)
{
   StringBuilder fileInformation = new StringBuilder(500);

   // Get the file name
   fileInformation.AppendLine("File Name: " + myDefinitionFile.Filename);

   // Iterate the Definition groups of this file
   foreach (Autodesk.Revit.Parameters.DefinitionGroup myGroup in myDefinitionFile.Groups)
   {
      // Get the group name
      fileInformation.AppendLine("Group Name: " + myGroup.Name);

      // Iterate the difinitions
      foreach (Autodesk.Revit.Parameters.Definition definition in myGroup.Definitions)
      {
         // Get definition name
         fileInformation.AppendLine("Definition Name: " + definition.Name);
      }
   }
   MessageBox.Show(fileInformation.ToString(), "Revit");
}
```

### 18.2.3 Change the Parameter Definition Owner Group

The following sample shows how to change the parameter definition group owner.

**Code Region 4-SharedParameter**

```
private void ReadEditExternalParam(Autodesk.Revit.Parameters.DefinitionFile file)
{
   // Get ExternalDefinition from shared parameter file
   Autodesk.Revit.Parameters.DefinitionGroups myGroups = file.Groups;
   Autodesk.Revit.Parameters.DefinitionGroup myGroup = myGroups.get_Item("MyGroup");
   if (null == myGroup)
      return;

   Autodesk.Revit.Parameters.Definitions myDefinitions = myGroup.Definitions;
```

```
   Autodesk.Revit.Parameters.ExternalDefinition myExtDef =
      myDefinitions.get_Item("MyParam") as Autodesk.Revit.Parameters.ExternalDefinition;

   if (null == myExtDef)
      return;

   // Change the OwnerGroup of the ExternalDefinition
   Autodesk.Revit.Parameters.DefinitionGroup anotherGroup
     = myGroups.get_Item("AnotherGroup");
   if (null == anotherGroup)
      return;
   myExtDef.OwnerGroup = anotherGroup;
}
```

## 18.3 Binding

In the add-in code, complete the following steps to bind a specific parameter:

1. Use an InstanceBinding or a TypeBinding object to create a new Binding object that includes the categories to which the parameter is bound.
2. Add the binding and definition to the document using the Document.ParameterBindings object.

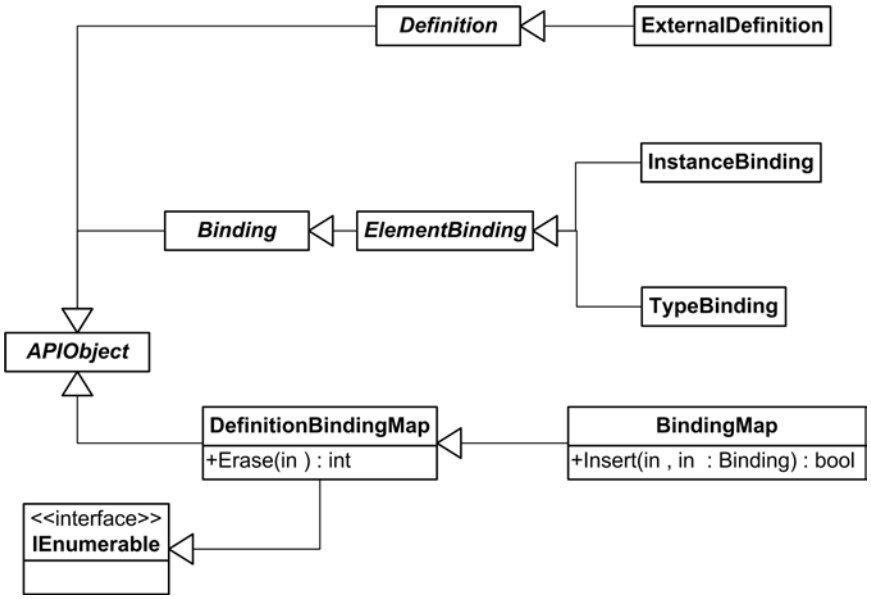



**Figure 188: Binding diagram**

The following list provides more information about the classes and methods in the previous diagram.

- Autodesk.Revit.Parameters.BindingMap Class - The BindingMap object is retrieved from the Document.ParameterBindings property.
  - Parameter binding connects a parameter definition to elements within one or more categories.
  - The map is used to interrogate existing bindings as well as generate new parameter bindings using the Insert method.

- Parameters.BindingMap.Insert (Definition, Binding) Method - The binding object type dictates whether the parameter is bound to all instances or just types.
  - A parameter definition cannot be bound to both instances and types.
  - If the parameter binding exists, the method returns false.

### 18.3.1 Type Binding

The Autodesk.Revit.Parameters.TypeBinding objects are used to bind a property to a Revit type, such as a wall type. It differs from Instance bindings in that the property is shared by all instances identified in type binding. Changing the parameter for one type affects all instances of the same type.

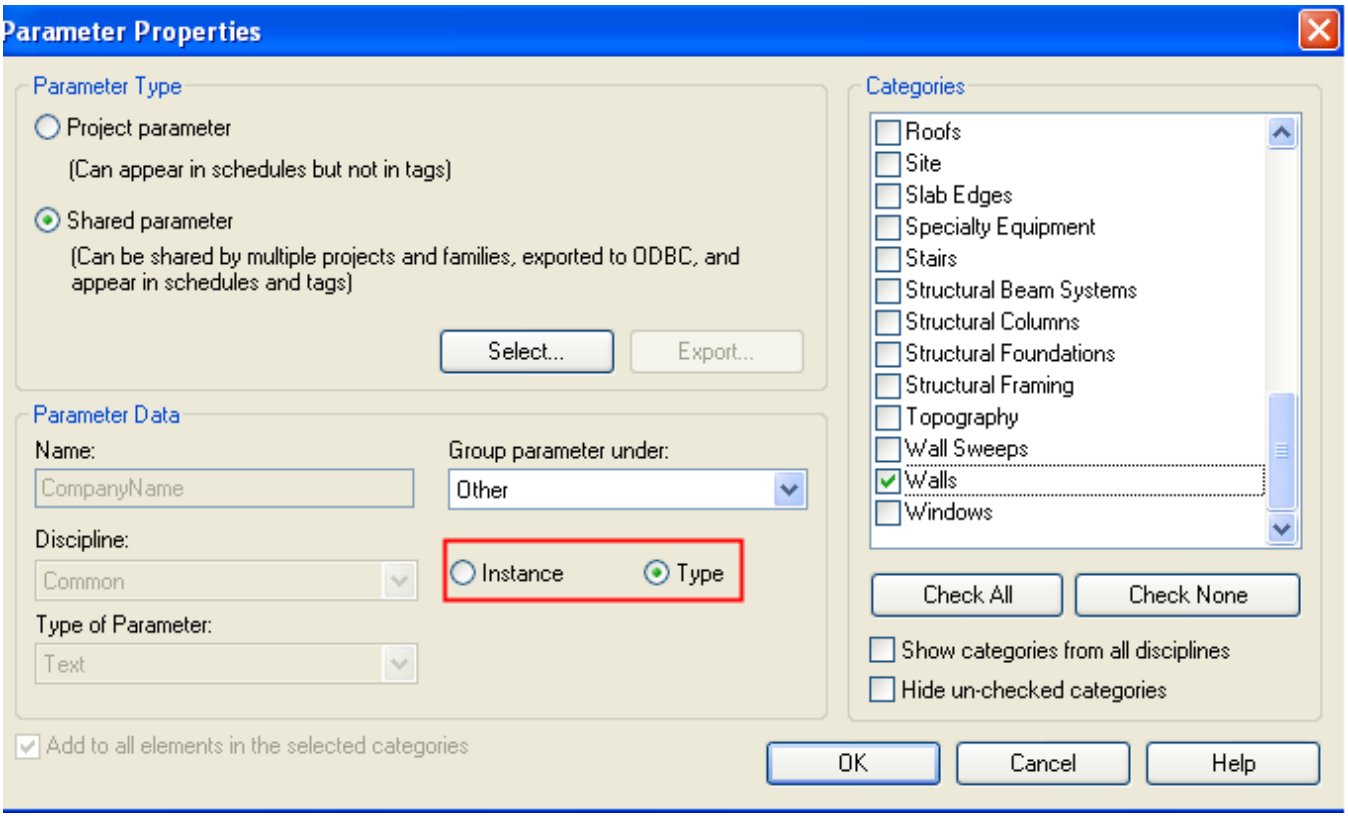



**Figure 189: Parameter Properties dialog box Type Binding**

The following code segment demonstrates how to add parameter definitions using a shared parameter file. The following code performs the same actions as using the dialog box in the previous picture. Parameter definitions are created in the following order:

1. A shared parameter file is created.
2. A definition group and a parameter definition are created for the Walls type.
3. The definition is bound to the wall type parameter in the current document based on the wall category.

**Code Region 5-SharedParameter**

```
// Create the CompanyName parameter in the definition file and add it
// to wall type in the current document
public bool SetNewParameterToTypeWall(
  Autodesk.Revit.Application app, DefinitionFile myDefinitionFile)
{
```

```
   // Create a new group in the shared parameters file
   DefinitionGroups myGroups = myDefinitionFile.Groups;
   DefinitionGroup myGroup = myGroups.Create("MyParameters");

   // Create a type definition
   Definition myDefinition_CompanyName =
      myGroup.Definitions.Create("CompanyName", ParameterType.Text);

   // Ccreate a category set and insert category of wall to it
   CategorySet myCategories = app.Create.NewCategorySet();
   // Use BuiltInCategory to get category of wall
   Category myCategory = app.ActiveDocument.Settings.Categories.get_Item(
     BuiltInCategory.OST_Walls);
   myCategories.Insert(myCategory);

   //Create an object of TypeBinding according to the Categories
   TypeBinding typeBinding = app.Create.NewTypeBinding(myCategories);

   // Get the BingdingMap of current document.
   BindingMap bindingMap = app.ActiveDocument.ParameterBindings;

   // Bind the definitions to the document
   bool typeBindOK = bindingMap.Insert(myDefinition_CompanyName, typeBinding,
     BuiltInParameterGroup.PG_TEXT);
   return typeBindOK;
}
```

### 18.3.2 Instance Binding

The Autodesk.Revit.Parameters.InstanceBinding object indicates binding between a parameter definition and a parameter in certain category instances. The following diagram illustrates Instance Binding in the Walls category.

Once bound, the parameter appears in all property dialog boxes for the instance. Changing the parameter in any one instance does not change the value in any other instance.

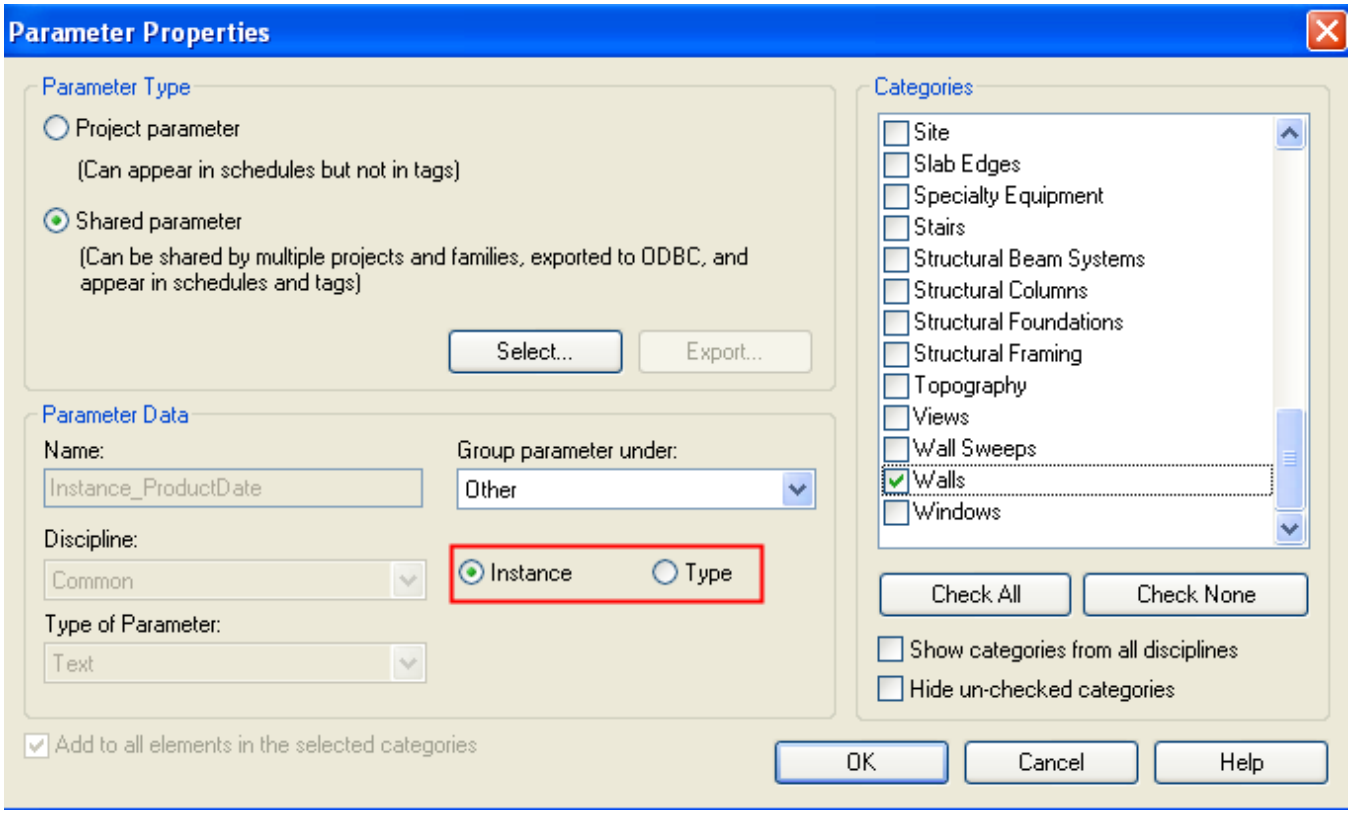



**Figure 190: Parameter Properties dialog box Instance Binding**

The following code sample demonstrates how to add parameter definitions using a shared parameter file. Parameter definitions are added in the following order:

1. A shared parameter file is created
2. A definition group and a definition for all Walls instances is created
3. Definitions are bound to each wall instance parameter in the current document based on the wall category.

**Code Region 6-SharedParameter**

```
// Create the Instance_ProductDate parameter in definition file and add
//it to the wall type instance in current document
public bool SetNewParameterToInsanceWall(
  Autodesk.Revit.Application app, DefinitionFile myDefinitionFile)
{
   // create a new group in the shared parameters file
   DefinitionGroups myGroups = myDefinitionFile.Groups;
   DefinitionGroup myGroup = myGroups.Create("MyParameters1");

   // create an instance definition in definition group MyParameters
   Definition myDefinition_ProductDate =
      myGroup.Definitions.Create("Instance_ProductDate", ParameterType.Text);

   // create a category set and insert category of wall to it
   CategorySet myCategories = app.Create.NewCategorySet();
   // use BuiltInCategory to get category of wall
```

```
   Category myCategory = app.ActiveDocument.Settings.Categories.get_Item(
     BuiltInCategory.OST_Walls);
   myCategories.Insert(myCategory);

   //Create an instance of InstanceBinding
   InstanceBinding instanceBinding = app.Create.NewInstanceBinding(myCategories);

   // Get the BingdingMap of current document.
   BindingMap bindingMap = app.ActiveDocument.ParameterBindings;

   // Bind the definitions to the document
   bool instanceBindOK = bindingMap.Insert(myDefinition_ProductDate,
    instanceBinding, BuiltInParameterGroup.PG_TEXT);
   return instanceBindOK;
}
```

# 19 Transaction

Transaction means enclosing multiple operations as one atomic operation. The following rules apply to transactions:

- A transaction is successful if and only if all operations within the transaction execute successfully.
- Any operation that fails causes the entire transaction to be aborted.
- If a transaction is aborted, all operations within that transaction are cancelled.

With few exceptions, all changes in persistent data in the document are grouped into Transactions. A Transaction corresponds to a single Undo action which is the main use of transactions in Revit. Another use is to trigger a set of operations that execute after a transaction is completed.

This chapter introduces the two uses for Transaction and the limits that you must consider.

## 19.1 Usage

The basic function of the transaction is to wrap a set of operations that can be transacted.

- The Document.BeginTransaction method starts the transaction
- The Document.EndTransaction method finishes the transaction when all operations are successful.
- The Document.AbortTransaction method rollbacks the whole transaction if an exception or a required abort operation occurs.

### 19.1.1 Atomic User Actions

The main transaction function in Revit is to initiate atomic user actions. All changes to persistent data are grouped into Transactions. A Transaction corresponds to a single undo action. Some changes affecting the display but not the model content can persist after saving the file but these situations are not managed by transactions. Transient Elements temporarily added to the document by editors are not persistent data. Unless the elements become non-transient, their addition and removal is not managed by transactions.

Examples of atomic user actions include the following:

- Create a wall, door, window, and so on.
- Cut/copy/paste
- Move/Rotate
- Delete
- Choose OK or Apply in a Properties dialog box.

Examples of non-atomic actions include the following:

- Press the left mouse button to start a pan. (A display change.)
- Create a poly-wall. (More than one transaction.)
- Drag the end of a wall when creating a wall command. (Less than one transaction.)
- Select (Without persistent effect.)

A transaction typically contains several smaller changes, including a direct user change and the resultant effects of regeneration. However, reversing a subset of these can leave the document in an inconsistent state. Therefore, changes are grouped and rollback to the document state is

permitted only to certain defined points in the change history. The following sample program demonstrates the transaction API function:

**Code Region 1 - Transaction**

```
// Autodesk.Revit.Application app = commandData.Application;
Autodesk.Revit.Document doc = app.ActiveDocument;

// Define the creator
Autodesk.Revit.Creation.Application appCreator = app.Create;
Autodesk.Revit.Creation.Document docCreator = doc.Create;

// Create a geometry line in revit application
XYZ Point1 = new XYZ(0, 0, 0);
XYZ Point2 = new XYZ(10, 0, 0);
XYZ Point3 = new XYZ(10, 10, 0);
XYZ Point4 = new XYZ(0, 10, 0);

// Transient Elements,so they are not managed by transactions
doc.BeginTransaction();
Line geomLine1 = appCreator.NewLine(ref Point1, ref Point2, true);
Line geomLine2 = appCreator.NewLine(ref Point4, ref Point3, true);
Line geomLine3 = appCreator.NewLine(ref Point1, ref Point4, true);

// Adding the following clause will not change the result
// revit.ActiveDocument.AbortTransaction();
doc.EndTransaction();

// Create a geometry plane in revit application
XYZ origin = new XYZ(0, 0, 0);
XYZ normal = new XYZ(1, 1, 0);
Plane geomPlane = appCreator.NewPlane(ref normal, ref origin);

// Create a sketch plane in current document
SketchPlane sketch = docCreator.NewSketchPlane(geomPlane);

// non-transient elements,so they are grouped into transaction as atomic user actions
doc.BeginTransaction();
// Create a ModelLine element using the created geometry line and sketch plane
ModelLine line1 = docCreator.NewModelCurve(geomLine1, sketch) as ModelLine;
ModelLine line2 = docCreator.NewModelCurve(geomLine2, sketch) as ModelLine;
ModelLine line3 = docCreator.NewModelCurve(geomLine3, sketch) as ModelLine;

if(DialogResult.OK == MessageBox.Show("Click OK to call EndTransaction, otherwise
  AbortTransaction.",  "Revit", MessageBoxButtons.OKCancel))
{
   doc.EndTransaction();
```

```
}
else
{
   doc.AbortTransaction();//A
}

//… other operations
```

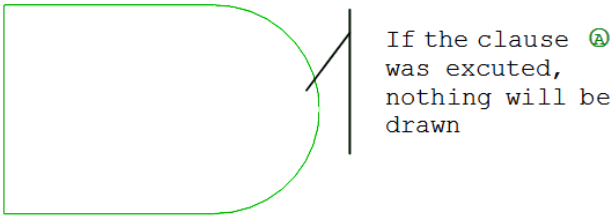



**Figure 191: Transaction result**

The transaction timeline for this sample is as follows:

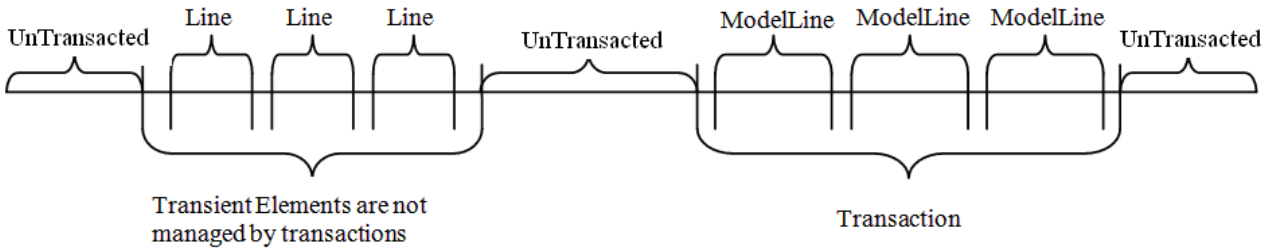



**Figure 192: Transaction timeline**

### 19.1.2 Get Created Element Geometry and AnalyticalModel

Another transaction function in Revit triggers operations that only execute after a transaction is completed. For example, if you create a wall then try to get the Geometry property, the property returns nothing.

It is because some operations cannot be executed before a transaction is completed. Essentially, an External Command is such a transaction. But it is also possible to use internal transaction to group creation events.

Using the AnalyticalModel property is the same as using Geometry. For more details about the AnalyticalModel property, refer to the AnalyticalModel section in the Revit Structure chapter. For more details about the Geometry property, refer to the Geometry chapter.

The following sample program demonstrates the API transaction function:

**Code Region 2 - Transaction**

```
//Autodesk.Revit.Application app = commandData.Application;
//… other operations


// Define the creator
```

```
Autodesk.Revit.Creation.Document docCreator = app.ActiveDocument.Create;

// Build a location line for the wall creation
XYZ start = new XYZ(0, 0, 0);
XYZ end = new XYZ(10, 10, 0);
Autodesk.Revit.Geometry.Line geomLine =  app.Create.NewLineBound(ref start, ref end);

app.ActiveDocument.BeginTransaction();
Level wallLevel = GetLevelForWallCreation(app.ActiveDocument);
// Create a wall using the location line
Wall wall = docCreator.NewWall(geomLine, wallLevel, true);
app.ActiveDocument.EndTransaction();

// If no transaction to group the creation of wall as above, cannot get the value // of
Geometry and AnalyticalModel.
Autodesk.Revit.Geometry.Options options = app.Create.NewGeometryOptions();
Autodesk.Revit.Geometry.Element geoelem = wall.get_Geometry(options);
Autodesk.Revit.Structural.AnalyticalModel analyticalmodel = wall.AnalyticalModel;

//… other operations
```

The transaction timeline for this sample is as follows:

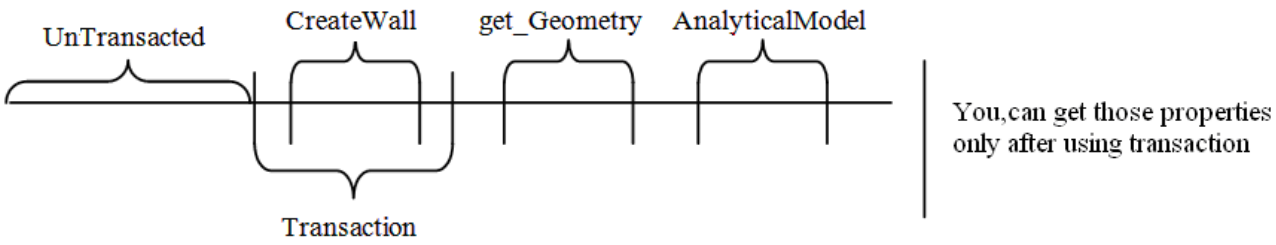



**Figure 193: Transaction timeline**

## 19.2 Boundaries

Because you are in charge of starting, ending, and aborting transactions, it is important to be aware of transaction boundaries. A transaction boundary is the period between starting and ending or aborting a transaction.

It is best to confine your boundary to a small scope. For example, if you start a transaction in a function, try to end or abort it before you return from that function since you might not have knowledge of the transaction outside of the function. It is not necessary to follow this rule if you want to maintain global managers for your transaction activity, but you should be responsible for aborting or ending all transactions you start.

The following sample program demonstrates transaction boundaries:

**Code Region 3 - Transaction**

```
//… other operations

try
{
   app.ActiveDocument.BeginTransaction();
   //… Some operation

   // End transaction before return from function
   // Rules in local function is the same
   app.ActiveDocument.EndTransaction();
   return IExternalCommand.Result.Succeeded;
}
catch(Exception ex)
{
   //… Some operation

   // Abort transaction when exception occured
   app.ActiveDocument.AbortTransaction();
   return IExternalCommand.Result.Failed;
}

//… other operations
```

Most applications use transaction management for operations on objects. The operations are committed at the end of the outermost transaction.

- A Revit command boundary is the boundary of your transaction as far as you can stretch.
- No active transactions exist when a command ends. If it does, Revit aborts (or crashes).

Generally speaking, it is a good idea to start a transaction when one of your functions is invoked and end the transaction when you return from that function. Most actions in Revit refer to this idea using the Document.BeginTransaction method and Document.EndTransaction method.

# 20 Revit Architecture

This chapter covers API functionality that is specific to Revit Architecture, namely:

- Functionality related to rooms (Element.Room, RoomTag, etc.)
- Limited energy analysis functions

**Note:** Currently, PlanTopology and PlanCircuit are used to create rooms.

## 20.1 Room

The following sections cover information about the room class, its parameters, and how to use the room class in the API.

### 20.1.1 Room, Area, and Tags

Room class is used to represent rooms drawn in the plan view. However, it also represents elements such as room schedule and area. The properties and create function for different rooms, areas, and their corresponding tags in the API are listed in the following table:

| Element | Class | Category | Boundary | Location | Can Create |
|---|---|---|---|---|---|
| Room in Plan View | Room | OST_Rooms | Has if in an enclosed region | LocationPoint | NewRoom and NewRooms except for NewRoom(Phase) |
| Room in Schedule View | Room | OST_Rooms | Null | Null | NewRoom(Phase) |
| Area | Room | OST_Areas | Always has | LocationPoint | No |
| Room Tag | RoomTag | OST_RoomTags | | LocationPoint | Creation.Document. NewRoomTag |
| Area Tag | FamilySymbol | OST_AreaTags | | LocationPoint | No |

**Table 43: Room, Area, and Tags relationship**

**Note**: Room.Name is the combination of the room name and room number. Use the ROOM_NAME BuiltInParameter to get the room name. In the following picture, the Room.Name returns Room 9.

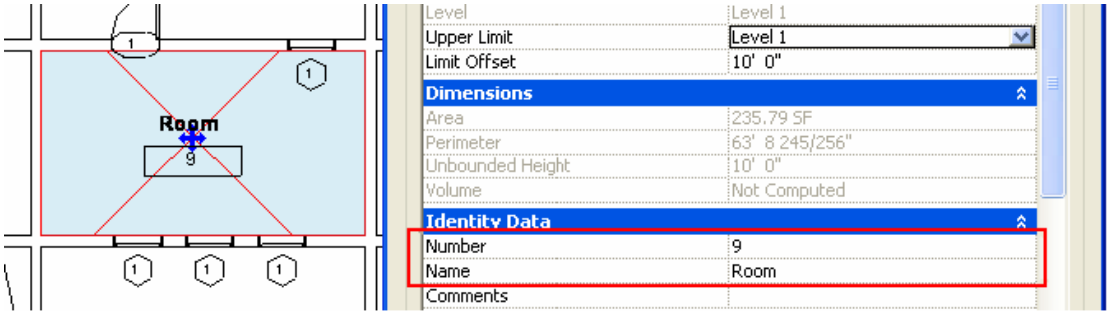



**Figure 194: Room name and number**

**Note**: As an Annotation Element, the specific view is available using RoomTag.View. Never try to set the RoomTag.Name property because the name is automatically assigned; otherwise an exception is thrown.

### 20.1.2 Create

The following code illustrates the simplest way to create a room at a certain point in a specific level:

**Code Region 1- Revit Architecture**

```
// Level level = getLevel();
UV uv = new UV(0, 0);
Room room = document.Create.NewRoom(level, ref uv);
RoomTag roomTag = document.Create.NewRoomTag(room, ref uv);
```

Rooms can be created in a room schedule then inserted into a plan circuit.

- The Document.NewRoom(Phase) method is used to create a new room and insert it into an existing schedule. Make sure the room schedule exists or create a room schedule in the specified phase before you make the call.
- The Document.NewRoom(Room room, PlanCircuit circuit) method is used to create a room from a schedule room and a PlanCircuit.
  - The input room must exist only in the room schedule meaning that it does not display in any plan view.
  - After invoking the method, a model room with the same name and number is created in the view where the PlanCircuit is located.

For more details about PlanCircuit, see the Plan Topology section in this chapter.

The following code illustrates the entire process:

**Code Region 2- Revit Architecture**

```
//Phase newConstructionPhase = GetPhase();
Debug.Assert(newConstructionPhase != null && planCircuit != null);
Room newScheduleRoom = document.Create.NewRoom(newConstructionPhase);
string newRoomNumber = "101";
string newRoomName = "Class Room 1";
newScheduleRoom.Name = newRoomName;
newScheduleRoom.Number = newRoomNumber;
Room newRoom2 = document.Create.NewRoom(newScheduleRoom, planCircuit);
Debug.Assert(newRoom2.Number == newRoomNumber && newRoomName ==
  newScheduleRoom.get_Parameter(BuiltInParameter.ROOM_NAME).AsString(   ));
// document.Create.NewRoom(newScheduleRoom, planCircuit);
// it must not display in any plan view to avoid exception
```

Create rooms in certain levels in batches. The NewRooms method can create rooms for every enclosed region in a given level.

### 20.1.3 Room Boundary

Rooms have boundaries that create an enclosed region where the room is located.

- Boundaries include the following elements:
  - Walls
  - Model lines

  - Columns
  - Roofs
  - And more.
- No more than two rooms can share the same boundary or a warning appears.

#### 20.1.3.1 Retrieve Boundaries

The key to recognizing the Room.Boundary is in the User Interface (UI). After selecting a wall, all the boundary segments are highlighted.

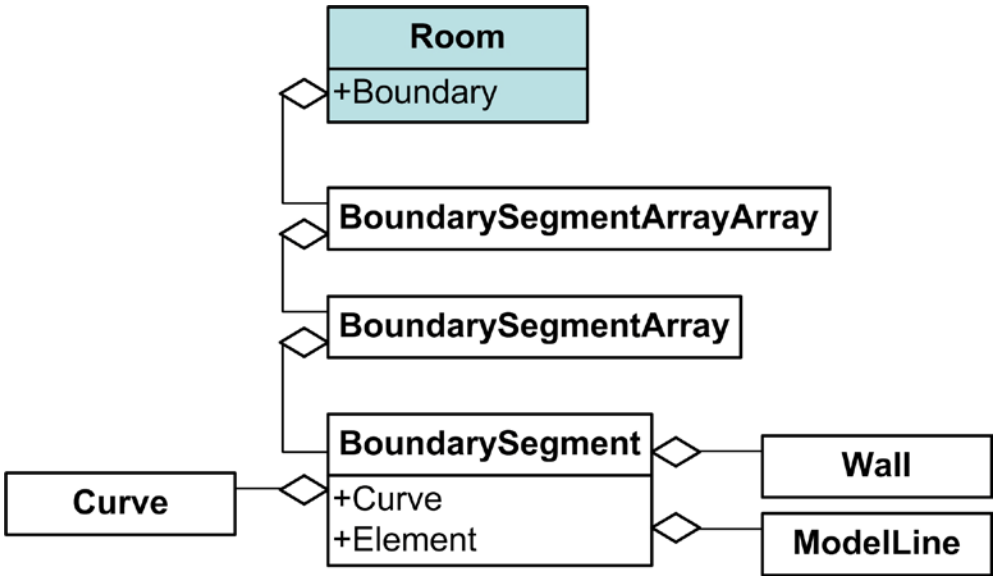



**Figure 195: Room boundary diagram**

In the following picture, notice the highlighted lines around the room.

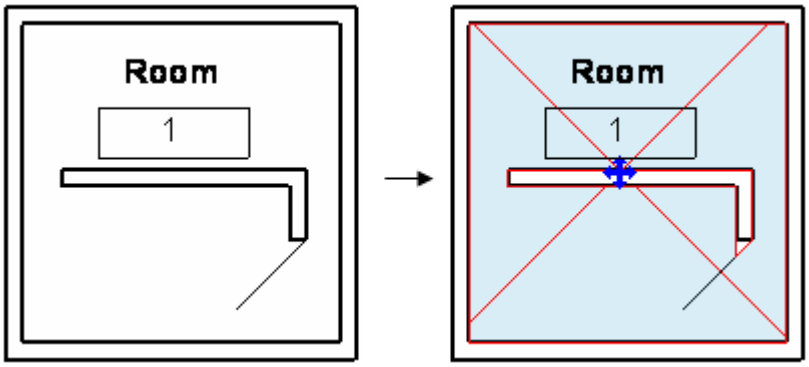



**Figure 196: Room boundary**

- Room.BoundaryArray is an array of BoundarySegment objects.
- Room.Boundary is null when the room is not in an enclosed region or only exists in the schedule.

The segment array size depends on the enclosed region topology. Each BoundarySegmentArray makes a circuit or a continuous line in which one segment joins the next. The following pictures provide several examples. In the following pictures, all walls are Room-Bounding and the model lines category is OST_AreaSeparationLines. If an element is not Room-Bounding, it is excluded from the elements to make the boundary.

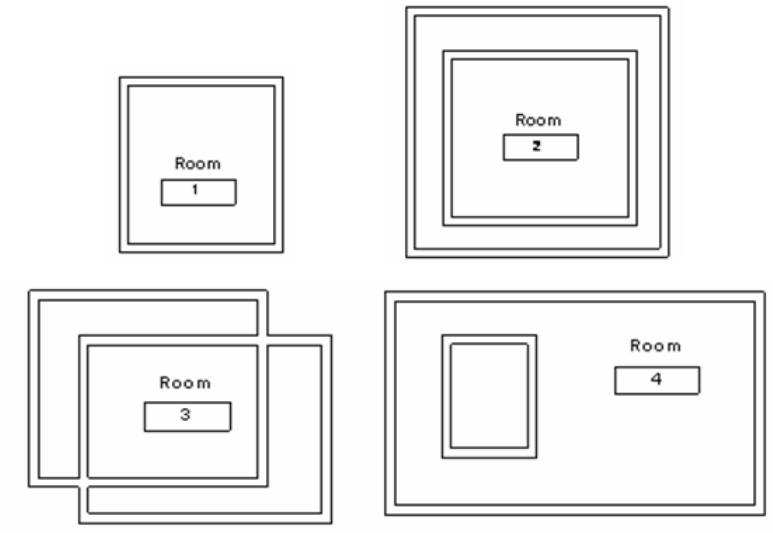



**Figure 197: Rooms 1, 2, 3, 4**

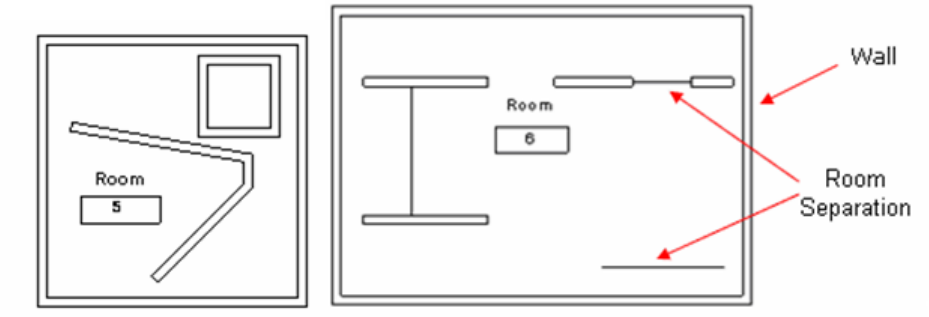



**Figure 198: Room 5, 6**

The following table provides the Room.Boundary.Size results for the previous rooms:

| Room | Room.Boundary.Size |
|---|---|
| Room 1<br>Room 2<br>Room 3 | 1 |
| Room 4 | 2 |
| Room 5<br>Room 6 | 3 |

**Table 44: Room.Boundary.Size**

**Note**: Walls joined by model lines are considered continuous line segments. Single model lines are ignored.

After getting BoundarySegmentArray, get the BoundarySegment by iterating the array.

### 20.1.3.2 BoundarySegment

The segments that make the region are represented by the BoundarySegment class; its Element property returns the corresponding element with the following conditions:

- For a ModelCurve element, the category must be BuiltInCategory.OST_AreaSeparationLines meaning that it represents a Room Separator.
- For other elements such as wall, column, and roof, if the element is a room boundary, the Room Bounding parameter (BuiltInParameter.WALL_ATTR_ROOM_BOUNDING) must be true as in the following picture.

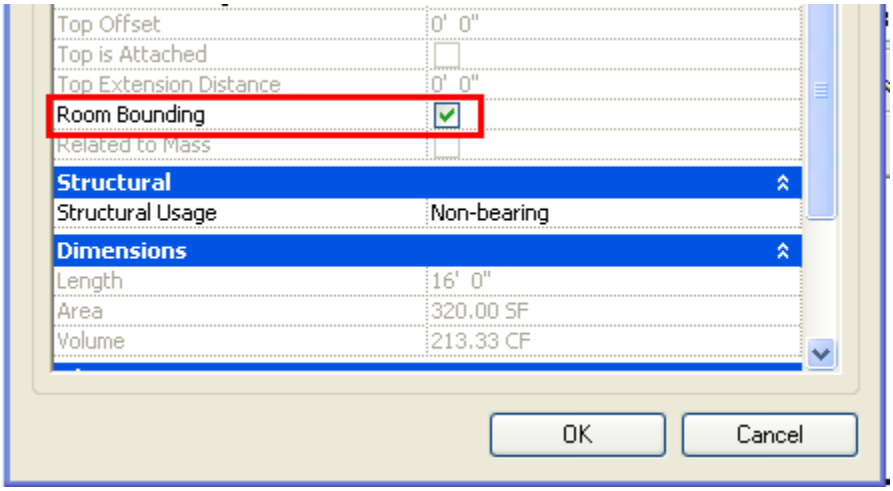



**Figure 199: Room Bounding property**

The WALL_ATTR_ROOM_BOUNDING BuiltInParameter is set through the Revit API:

**Code Region 3- Revit Architecture**

```
//Wall aWall = GetWall();
Parameter parameter = aWall.get_Parameter(BuiltInParameter.WALL_ATTR_ROOM_BOUNDING);
parameter.Set(1);   //set "Room Bounding" to true
parameter.Set(0);   //set "Room Bounding" to false
```

Notice how the roof forms the BoundarySegment for a room in the following pictures. The first picture shows Level 3 in the elevation view. The room is created in the Level 3 floor view. The latter two pictures show the boundary of the room and the house in 3D view.

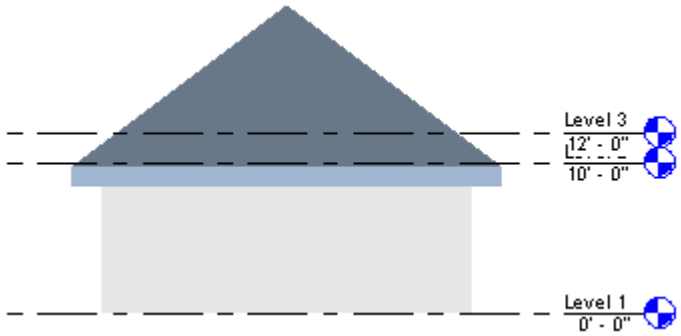



**Figure 200: Room created in level 3 view**

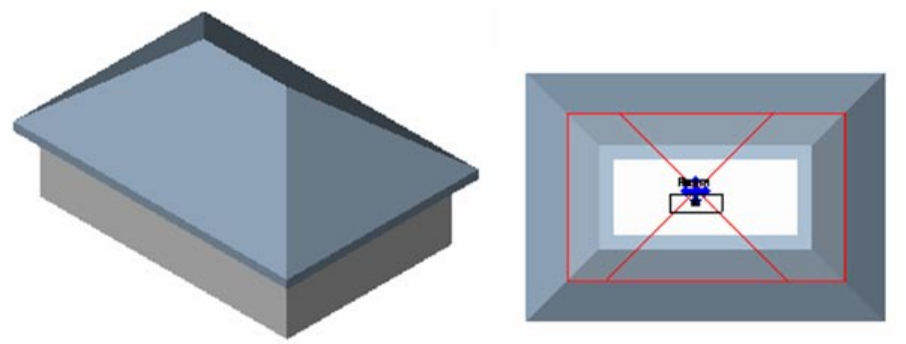


**Figure 201: Room boundary formed by roof**

The area boundary can only be a ModelCurve with the category Area Boundary (BuiltInCategory.OST_AreaSchemeLines) while the boundary of the displayed room can be walls and other elements.

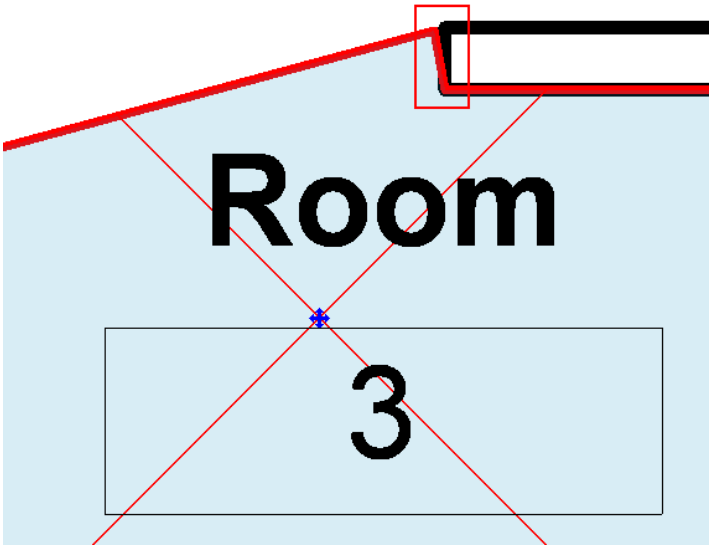



**Figure 202: Wall end edge**

If the BoundarySegment corresponds to the curve between the room separation and wall as the previous picture shows:

- The Element property returns null
- The Curve is not null.

### 20.1.3.3 Boundary and Transaction

When you call Room.Boundary after creating an Element using the API such as a wall, the wall can change the room boundary. You must make sure the data is updated.

The following illustrations show how the room changes after a wall is created using the Revit API.

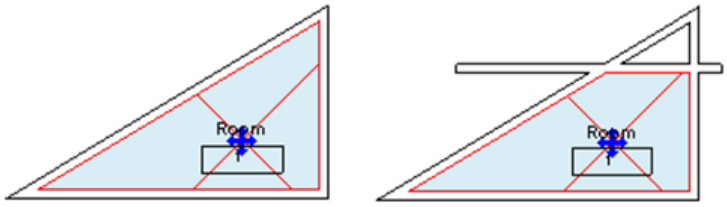


**Figure 203: Added wall changes the room boundary**

To update the room boundary data, use the transaction mechanism in the following code:

**Code Region 4- Revit Architecture**

```
Autodesk.Revit.Application app = commandData.Application;
Document doc = app.ActiveDocument;
Room room = GetRoomWithBoundary(doc);
Level aLevel = GetLevel(doc);

//Get the size before creating a wall
int size = room.Boundary.get_Item(0).Size;

//Prepare a line
XYZ startPos = new XYZ(-10, 0, 0);
XYZ endPos = new XYZ(10, 0, 0);
Line line = app.Create.NewLine(ref startPos, ref endPos, true);

//Create a new wall and enclose the creating into a single transaction
doc.BeginTransaction();
Wall wall = doc.Create.NewWall(line, aLevel, false);
doc.EndTransaction();

//Get the new size
size = room.Boundary.get_Item(0).Size;
```

For more details, see the Transaction chapter.

### 20.1.4 Plan Topology

The level plan that rooms lie in have a topology made by elements such as walls and room separators. The PlanTopology and PlanCircuit classes are used to present the level topology.

- Get the PlanTopology object from the Document object using the Level. In each plan view, there is one PlanTopology corresponding to every phase.
- The same condition applies to BoundarySegment, except room separators and Elements whose Room Bounding parameter is true can be a side (boundary) in the PlanCircuit.

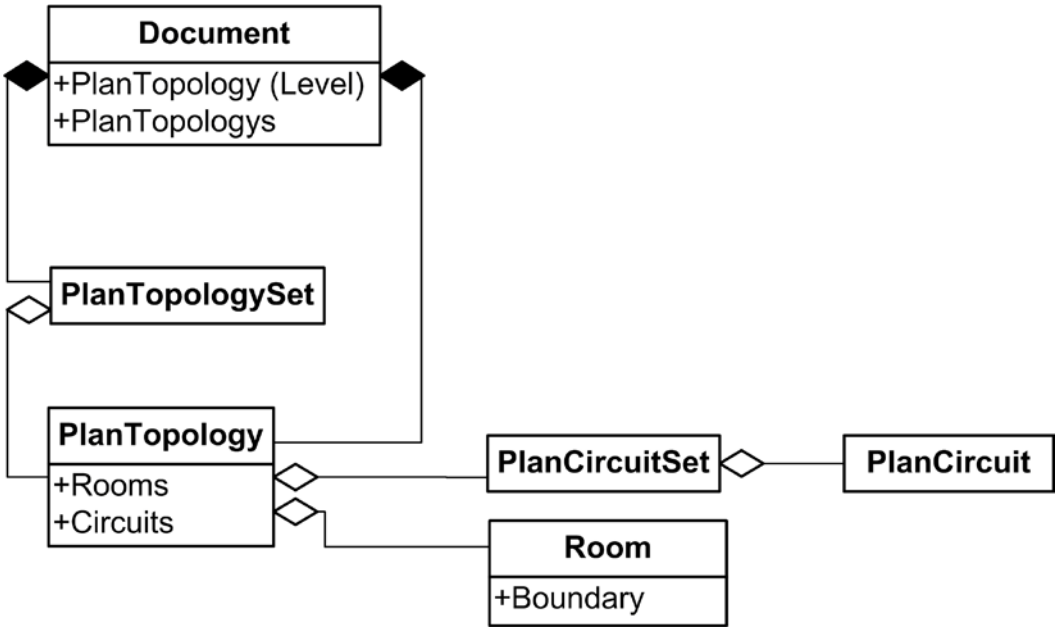



**Figure 204: Room and Plan Topology diagram**

The PlanCircuit.SideNum property returns the circuit side number, which is different from the BoundarySegmentArray.Size.

- The BoundarySegmentArray.Size recognizes the bottom wall as two walls if there is a branch on the wall.
- PlanCircuit.SideNum always sees the bottom wall in the picture as one regardless of the number of branches.

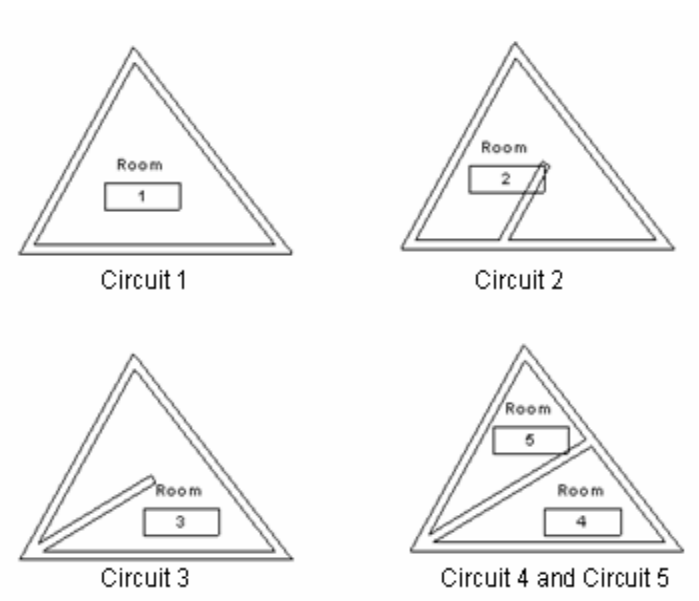



**Figure 205: Compare room boundary with PlanCircuit**

| Circuit | Circuit.SideNum | BoundarySegmentArray.Size for Room |
|---|---|---|
| Circuit 1 | 3 | 3 (Room1) |
| Circuit 2 | 4 +2 = 6 | 4 +3 = 7 (Room2) |

| Circuit | Circuit.SideNum | BoundarySegmentArray.Size for Room |
|---|---|---|
| Circuit 3 | 3 +2 = 5 | 3 +3 = 6 (Room3) |
| Circuit 4 | 3 | 3 (Room4) |
| Circuit 5 | 3 | 3 (Room5) |

**Table 45: Compare Room Boundary with PlanCircuit**

### 20.1.5 Room and FamilyInstance

Doors and Windows are special family instances related to Room. Only doors are discussed here since the only difference is that windows have no handle to flip.

The following characteristics apply to doors:

- Door elements can exist without a room.
- -In the API (and only in the API), a Door element has two additional properties that refer to the regions on the two opposite sides of a door: ToRoom and FromRoom
  - -If the region is a room, the property's value would be a Room element.
  - -If the region is not a room, the property will return null. Both properties may be null at the same time.
  - -The region on the side into which a door opens, will be ToRoom. The room on the other side will be FromRoom.
  - Both properties gets dynamically updated whenever the corresponding regions change.

In the following pictures, five doors are inserted into walls without flipping the facing. The table lists the FromRoom, ToRoom, and Room properties for each door. The Room property belongs to all Family Instances.

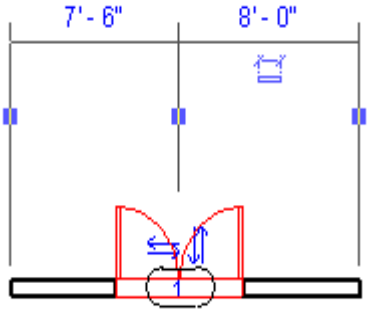



**Figure 206: Door 1**

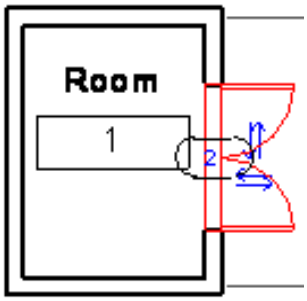



**Figure 207: Door 2**

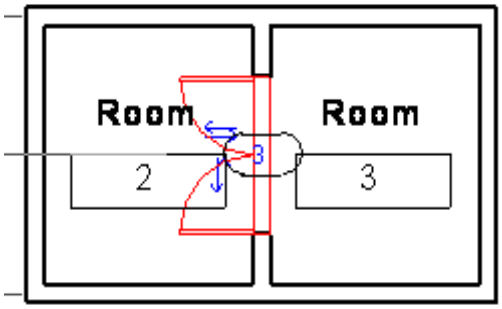



**Figure 208: Door 3**

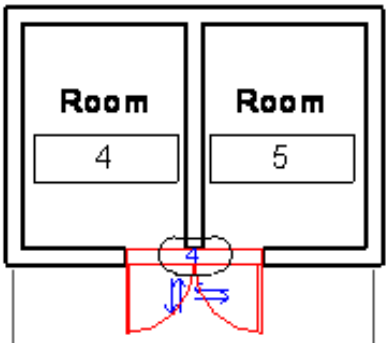



**Figure 209: Door 4**

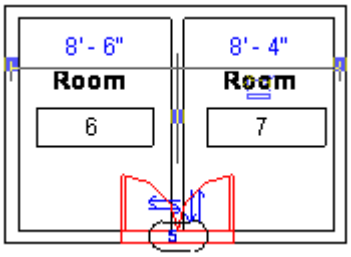



**Figure 210: Door 5**

| Door | FromRoom | ToRoom | Room |
|---|---|---|---|
| Door 1 | null | null | null |
| Door 2 | Room 1 | null | null |
| Door 3 | Room 3 | Room 2 | Room 2 |
| Door 4 | Room 4 | null | null |
| Door 5 | null | Room 6 | Room 6 |

**Table 46: Door Properties**

All family instances have the Room property, which is the room where an instance is located in the last project phase. Windows and doors face into a room. Change the room by flipping the door or window facing. For other kinds of instances, such as beams and columns, the Room is the room that has the same boundary as the instance.

The following code illustrates how to get the Room from the family instance. It is necessary to check if the result is null or not.

**Code Region 5- Revit Architecture**

```
Room room = familyInstance.Room;
room = familyInstance.FromRoom;  //for door and window family only
room = familyInstance.ToRoom;    //for door and window family only
if(null != room)
{
   //use the room...
}
```

## 20.2 Energy Data

The gbXMLParamElem object represents the gbXML Parameters in the Revit project. To view the parameters, from the menu, select Settings > Project Information…. The Project Information dialog box appears.

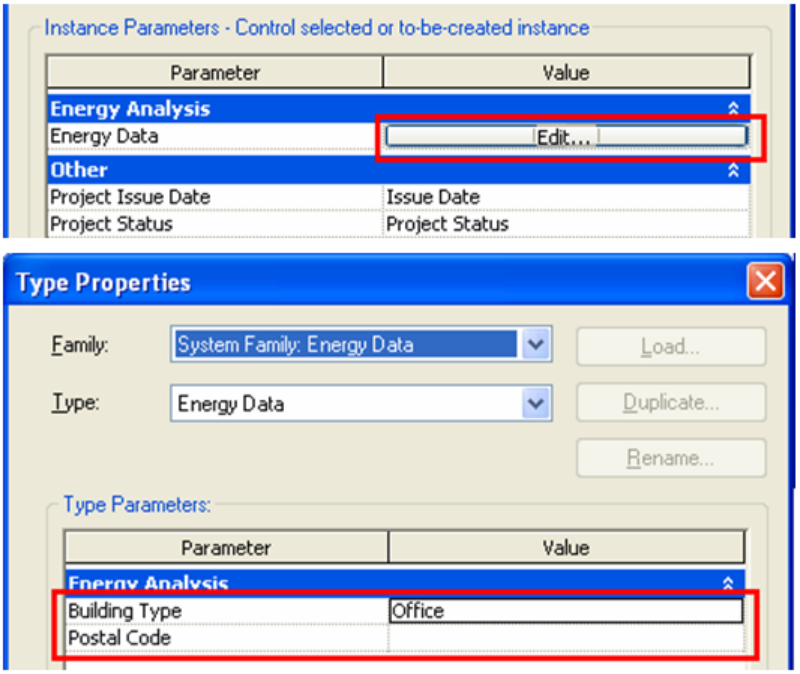



**Figure 211: gbXMLParamElem setting**

The gbXMLParamElem object is derived from the Element base object. It is unique in each project, similar to ProjectInformation. Though gbXMLParamElem is a subclass of the Element class, most of the members inherited from the Element return null or an empty set except for Name, Id, UniqueId, and Parameters. In the Revit English version, Name returns Energy Data with the following Parameters:

- The BuildingType property is used to get or set the Project Information Building Type.
- The ZIPCode property is used to get or set the Project Information ZIP Code.

The following code sample uses the gbXMLParamElem class. The result appears in a message box after invoking the command. The following sample illustrates how to create a new grid with a line.

**Code Region 6 - Revit Architecture**

```
public IExternalCommand.Result Execute(ExternalCommandData commandData,ref string message,
  ElementSet elements)
{
   Autodesk.Revit.Application app = commandData.Application;
```

```
   Document doc = app.ActiveDocument;
   // The gbXMLParamElem can be retrieved from
   // ProjectInformation only
   gbXMLParamElem xmlPara = doc.ProjectInformation.gbXMLSettings;
   if (null != xmlPara)
   {
      GetInfo_gbXMLParamElem(xmlPara);
   }
   return IExternalCommand.Result.Succeeded;
}
public void GetInfo_gbXMLParamElem(gbXMLParamElem xmlParamElem)
{
   string message = "gbXMLParamElem : ";
   message += "\nBuildingType : " + xmlParamElem.BuildingType;
   message += "\nZIPCode : " + xmlParamElem.ZIPCode;
   MessageBox.Show(message, "Revit", MessageBoxButtons.OK);
}
```

# 21 Revit Structure

Certain API features that only exist in Revit Structure products are discussed in the following sections:

- Structural Model Elements - Discusses specific Elements and their properties that only exist in the Revit Structure product.
- AnalyticalModel - Discusses analytical model-related classes such as AnalyticalModel, RigidLink, and SupportData.
- Loads - Discusses Load Settings and three kinds of Loads.
- Your Analysis Link - Provides suggestions for API users who want to link the Revit Structure product to certain Structural Analysis applications.

This chapter contains some advanced topics. If you are not familiar with the Revit API, read the basic chapters first, such as Getting Started, Elements Essentials, Parameter, and so on.

## 21.1 Structural Model Elements

Structural Model Elements are, literally, elements that support a structure such as columns, rebar, and so on. This section discusses how to manipulate these elements.

### 21.1.1 Column, Beam, and Brace

Currently Structural column, beam, and brace elements do not have a specific class such as the StructuralColumn class but they are in the FamilyInstance class form. Specific classes for these structural elements are expected in the next API release.

**Note:** Though the StructuralColumn, StructuralBeam, and StructuralBrace classes do not exist in the current API, they are used in this chapter to indicate the FamilyInstance objects corresponding to structural columns, beams, and braces.

Though Structural column, beam, and brace are all FamilyInstance objects in the API, they are distinguished by Category and StructuralType property.

**Code Region 1 - RevitStructure**

```
// FamilyInstance inst = GetFamilyInstance();
Category cate = inst.Category;
if (null == cate)
{
    return;
}
if (cate.Id.Value == (int)BuiltInCategory.OST_StructuralColumns)
{
    FamilyInstance aColumn = inst;
}
else if (cate.Id.Value == (int)BuiltInCategory.OST_StructuralFraming)
{
    if (StructuralType.Beam == inst.StructuralType)
    {
        FamilyInstance aBeam = inst;
    }
```

```
    else if (StructuralType.Brace == inst.StructuralType)
    {
        FamilyInstance aBrace = inst;
    }
}
```

You can filter out FamilySymbol objects corresponding to structural columns, beams, and braces in the same way. The category for Structural beams and braces is BuiltInCategory.OST_StructuralFraming.

**Code Region 2 - RevitStructure**

```
FamilySymbolSet columnTypes = new FamilySymbolSet();
FamilySymbolSet framingTypes = new FamilySymbolSet();
Document doc = commandData.Application.ActiveDocument;
ElementFilterIterator elemFilterItor = doc.get_Elements(typeof(Family));
elemFilterItor.Reset();
while (elemFilterItor.MoveNext())
{
    Family tmpFamily = elemFilterItor.Current as Family;
    Category category = tmpFamily.Category;
    if (null != category)
    {
        if ((int)BuiltInCategory.OST_StructuralColumns == category.Id.Value)
        {
            foreach (FamilySymbol tmpSymbol in tmpFamily.Symbols)
            {
               columnTypes.Insert(tmpSymbol);
            }
        }
        else if ((int)BuiltInCategory.OST_StructuralFraming == category.Id.Value)
        {
            foreach (FamilySymbol tmpSymbol in tmpFamily.Symbols)
            {
               framingTypes.Insert(tmpSymbol);
            }
        }
    }
}
```

### 21.1.2 AreaReinforcement and PathReinforcement:

Find the AreaReinforcementCurves for AreaReinforcement by iterating the Curves property which is an ElementArray. In edit mode, AreaReinforcementCurves are the purple curves (red when selected) in the Revit UI. From the Element Properties dialog box you can see AreaReinforcementCurve parameters. Parameters such as Hook Types and Orientation are editable only if the Override Area Reinforcement Setting parameter is true.

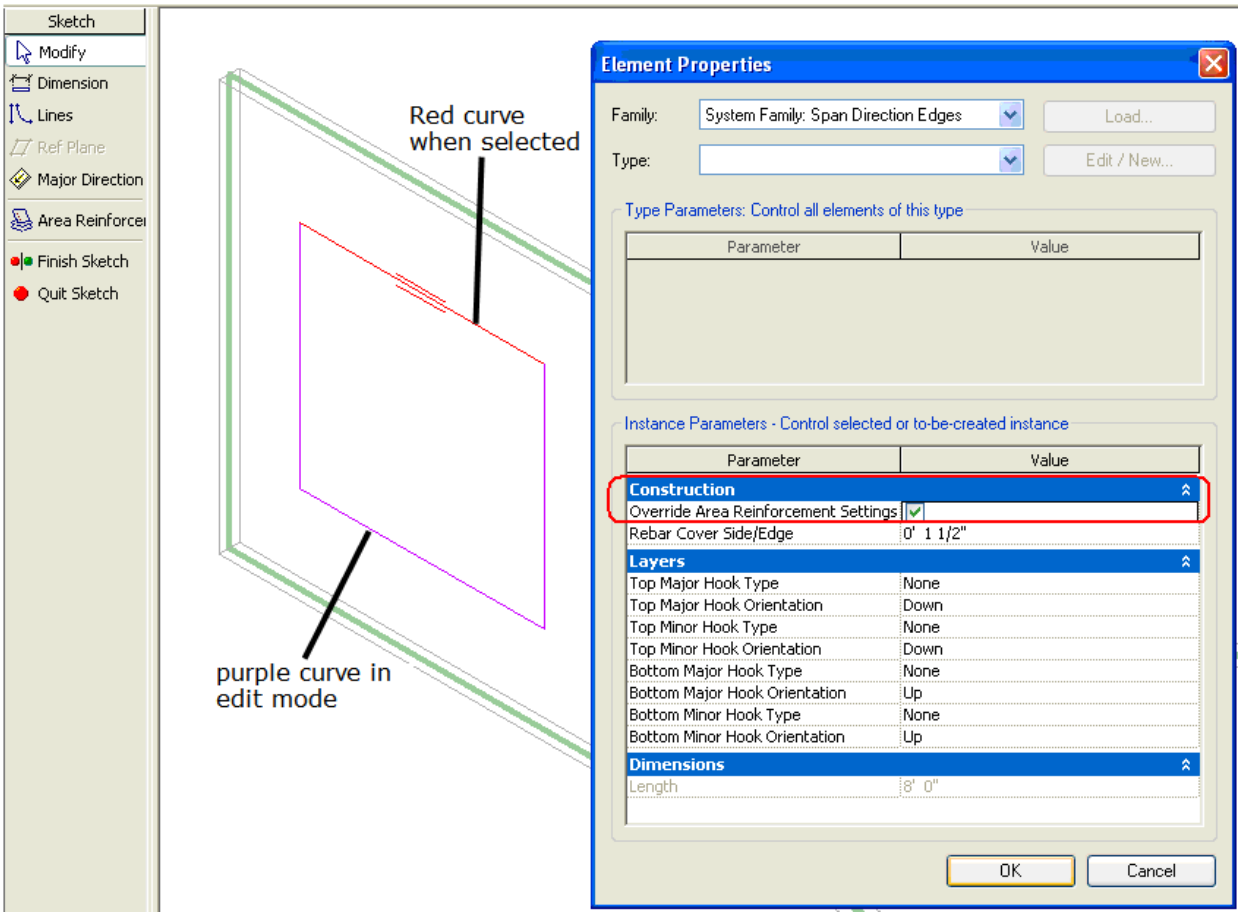



**Figure 212: AreaReinforcementCurve in edit mode**

Currently there is no way to control the direction except by using the NewAreaReinforcement method (the last XYZ direction input parameter).

```
public AreaReinforcement NewAreaReinforcement(
   AreaReinforcementType areaReinforcementType,
   Element host, CurveArray curves, XYZ direction);
```

Although the AreaReinforcement Curves property returns a set of AreaReinforcementCurves, PathReinforcementCurves returns a ModelCurve. Currently, there is no way to flip the PathReinforcement except by using the NewPathReinforcement method (the last input parameter).

```
public PathReinforcement NewPathReinforcement(
   PathReinforcementType pathReinforcementType,
   Element host, CurveArray curves, bool flip);
```

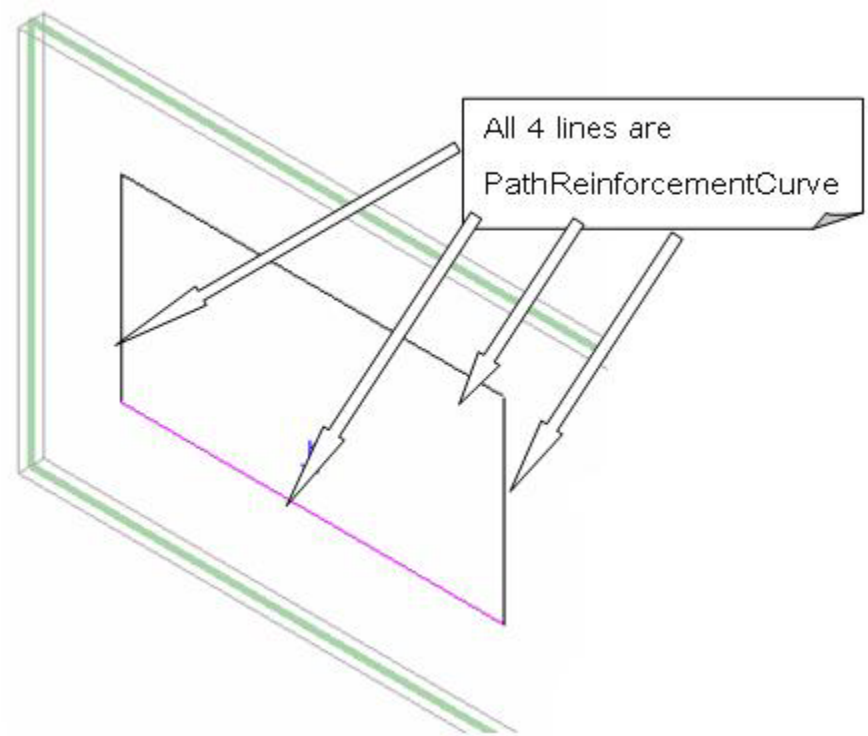



**Figure 213: PathReinforcement.PathReinforcementCurve in edit mode**

When using NewAreaReinforcement and NewPathReinforcement methods to create objects, you must decide on which host Element face it will lay. Currently AreaReinforcement and PathReinforcement are only created on the PlanarFace retrieved from the Wall or Floor object. After removing the faces from the Wall or Floor geometry, you can filter the PlanarFace out as follows:

- Downcast the Face to PlanarFace:

**Code Region 3 - RevitStructure**

```
// Get Face aPlanarFace
PlanarFace aPlanarFace = aFace as PlanarFace;
```

- If it is a PlanarFace, get its Normal and Origin:

**Code Region 4 - RevitStructure**

```
if (null != aPlanarFace)
{
    XYZ normal = aPlanarFace.Normal;
    XYZ origin = aPlanarFace.Origin;
}
```

- Filter out the right face based on normal and origin. For example:
  - For a general vertical Wall, the face is located using the following factors:

    The face is vertical; (normal.Z == 0.0)

    Parallel face directions are opposite:

(normal1.X = - normal2.X; normal1.Y = - normal2.Y)

Normal must be parallel to the location line.

– For a general Floor without slope, the factors are:

The face is horizontal; (normal.X == 0.0 && normal.Y ==0.0)

Judge the top and bottom face; (distinguish 2 faces by normal.Z)

For more details about retrieving an Element's Geometry, refer to the Geometry chapter.

### 21.1.3 BeamSystem

BeamSystem provides you with full access and edit ability. You can get and set all of its properties, such as BeamSystemType, BeamType, Direction, and Level. BeamSystem.Direction is not limited to one line of edges. It can be set to any XYZ coordinate on the same plane with the BeamSystem.

**Note:** You cannot change the StructuralBeam AnalyticalModel after the Elevation property is changed in the UI or by the API. In the following picture the analytical model lines stay in the original location after BeamSystem Elevation is changed to 10 feet.

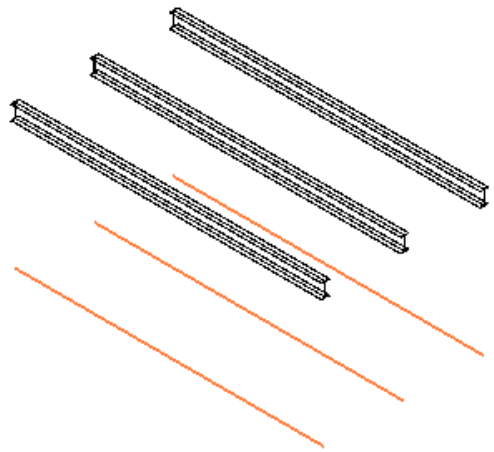


**Figure 214: Change BeamSystem elevation**

### 21.1.4 Rebar

When using the NewRebar method, the parameters RebarBarType and RebarHookType are available in the RebarBarTypes and RebarHookTypes property. The RebarBarTypes property and RebarHookTypes property are Document properties.

```
public Rebar NewRebar(RebarBarType rebarType,
RebarHookType startHook, RebarHookType endHook,
Element host, ref XYZ origin, ref XYZ norm,
    CurveArray curves, int startHookOrient, int endHookOrient);
```

The following code illustrates how to create Rebar with a specific layout.

**Code Region 5 - RevitStructure**

```
virtual public IExternalCommand.Result
Execute(ExternalCommandData commandData,
   ref String message, ElementSet elements)
```

```
{
   // …
   // Get necessary input parameters
   Rebar bar1 = document.Create.NewRebar(barType, hookType, hookType, host, ref origin, ref
  tmpPnt, curves, 0, 0);
   Rebar bar2 = document.Create.NewRebar(barType, hookType, hookType, host, ref origin, ref
  normal, curves, 0, 0);
   SetRebar(bar1);
   SetRebar(bar2);
   // …
}
void SetRebar(Rebar bar)
{
   Parameter paramLayout = bar.get_Parameter(BuiltInParameter.REBAR_ELEM_LAYOUT_RULE);
   paramLayout.Set(1);
   Parameter paramNum = bar.get_Parameter(BuiltInParameter.REBAR_ELEM_QUANTITY_OF_BARS);
   paramNum.Set(10);
   bar.ArrayLength = 1.5;
}
```

The following table lists the integer value for the Parameter REBAR_ELEM_LAYOUT_RULE:

| Value | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| Description | None | Fixed Number | Maximum Spacing | Number with Spacing | Minimum Clear Spacing |

**Table 47: Rebar Layout Rule**

In the NewRebar method input parameters, the following rules apply:

- All curves must lie on the same plane.
- If the input parameter, normal, is not vertical to the plane, the curves are projected to the plane that is vertical to the normal and contains the origin point.

In the following picture, the input curves are horizontal rectangles.

- The horizontal Rebar is created with vertical normal.
- The declining Rebar is created with the same curves and another normal parallel to the normal.
- The rectangle is projected to the plane vertical to the normal.

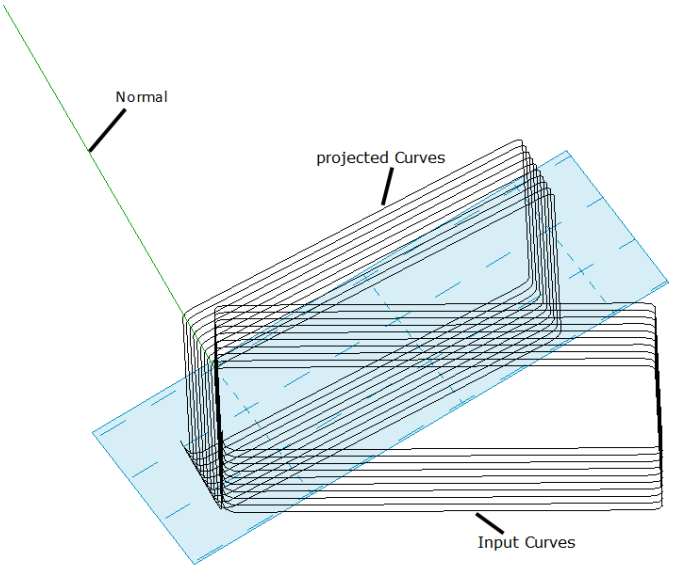



**Figure 215: Normal is not vertical to curves in NewRebar**

### 21.1.5 BoundaryConditions

There are three types of BoundaryConditions:

- Point
- Curve
- Area

The type and pertinent geometry information is retrieved using the following code:

**Code Region 6 - RevitStructure**

```
//BoundaryConditions aBC = GetBoundaryConditions();
Parameter param = aBC.get_Parameter(BuiltInParameter.BOUNDARY_CONDITIONS_TYPE);
switch (param.AsInteger())
{
    case 0:
        XYZ point = aBC.Point;
        break;
    case 1:
        Curve curve = aBC.get_Curve(0);
        break;
    case 2:
        CurveArray profile = new CurveArray();
        for (int i = 0; i < aBC.NumCurves; i++)
        {
            profile.Append(aBC.get_Curve(i));
        }
        break;
    default:
        break;
}
```

### 21.1.6 Other Structural Elements

Some Element derived classes exist in Revit Architecture and Revit Structure products. In this section, methods specific to Revit Structure are introduced. For more information about these classes, see the corresponding parts in the Host Elements and Opening and Component Elements chapters.

#### 21.1.6.1 Slab

Both Slab (Structural Floor) and Slab Foundation are represented by the Floor class and are distinguished by the IsFoundationSlab property.

The Slab Span Directions are represented by the IndependentTag class in the API and are available as follows:

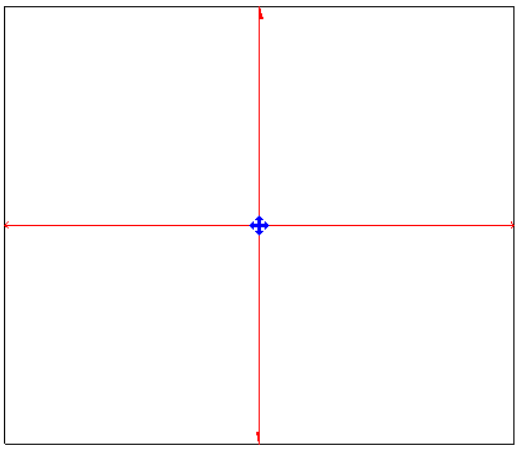


**Figure 216: Slab span directions**

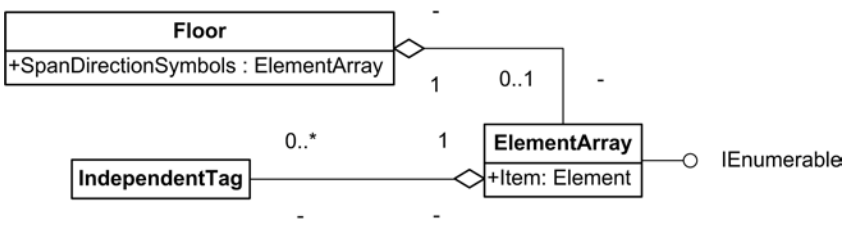



**Figure 217: SpanDirectionSymbols diagram**

When using NewSlab to create a Slab, Span Directions are not automatically created. Currently there is also no way to create it directly.

The Slab compound structure layer Structural Deck properties are exposed by the following properties:

- CompoundStructuralLayer DeckUsage
- DeckProfile

The properties are outlined in the following dialog box:

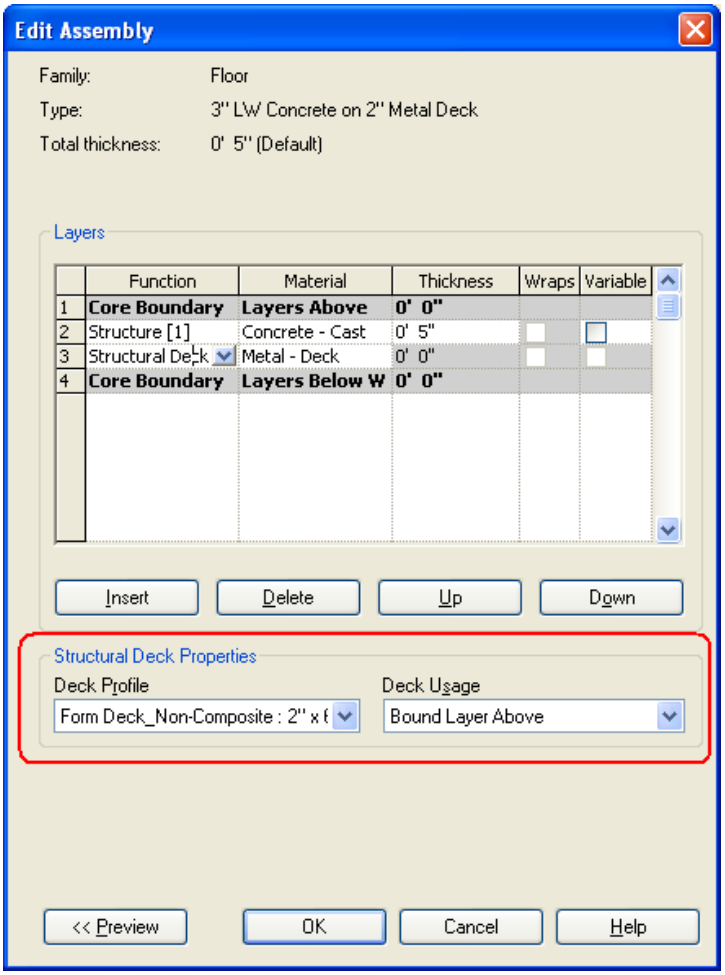



**Figure 218: Floor CompoundStructuralLayer unsupported functions**

## 21.2 AnalyticalModel

In Revit Structure, an analytical model is the engineering description of a structural physical model.

The following structural elements have elemental analytical models:

- Column
- Beam
- Slab
- Brace
- Wall

In the API, an Element object's AnalyticalModel type is determined by the class type, Category, and other properties. Depending on the element's family, the AnalyticalModel may not exist.

If the AnlyticalModel value does not apply to an element's family, the AnalyticalModel property is null. Check the value before using this class. The following table lists the structural elements and AnalyticalModel types in the API:

| Object | Element Type | Category (BuiltInCategory) | AnalyticalModel Type |
|---|---|---|---|
| Continuous Wall Foundation | ContFooting | OST_StructuralFoundation | AnalyticalModel3D |
| Structural Wall | Wall | OST_Walls | AnalyticalModelWall |
| Slab | Floor | OST_Floors | AnalyticalModelFloor |
| Beam | StructuralBeam | OST_StructuralFraming | AnalyticalModelFrame |
| Brace | StructuralBrace | OST_StructuralFraming | AnalyticalModelFrame |
| Column | StructuralColumn | OST_StructuralColumns | AnalyticalModelFrame |
| Isolated Footing | FamilyInstance | OST_StructuralFoundation | AnalyticalModelLocation |
| In-Place Member | FamilyInstance | Corresponding Category | AnalyticalModel3D |

**Table 48: Elements and Corresponding AnalyticalModel Type**

Use the following code to get the proper AnalyticalModel for different elements:

**Code Region 7 - RevitStructure**

```
//Autodesk.Revit.Element elem = GetElement();
Wall aWall = elem as Wall;
if (null != aWall)
{
AnalyticalModelWall modelWall
= aWall.AnalyticalModel as AnalyticalModelWall;
}
Floor aFloor = elem as Floor;
if (null != aFloor)
{
  AnalyticalModelFloor modelFloor = aFloor.AnalyticalModel as AnalyticalModelFloor;
}
ContFooting aContFooting = elem as ContFooting;
if (null != aContFooting)
{
   AnalyticalModel3D model3D =
aContFooting.AnalyticalModel as AnalyticalModel3D;
}
FamilyInstance famInst = elem as FamilyInstance;
if (null != famInst)
{
   Category cate = famInst.Category;
   if (null != cate)
   {
      if (cate.Id.Value == (int)BuiltInCategory.OST_StructuralColumns
          || cate.Id.Value == (int)BuiltInCategory.OST_StructuralFraming)
      {
         AnalyticalModelFrame modelFrame =
```

```
famInst.AnalyticalModel as AnalyticalModelFrame;
AnalyticalModelProfile modelProfile = modelFrame.Profile;
        }
if (cate.Id.Value == (int)BuiltInCategory.OST_StructuralFoundation)
      {
         AnalyticalModelLocation modelLocation = famInst.AnalyticalModel as
         AnalyticalModelLocation;
      }
   }
}
```

- AnalyticalModelProfile objects are available in the AnalyticalModelFrame Profile property; you can get it from the AnalyticalModel property directly.
- The In-place member AnalyticalModel property returns an AnalyticalModel3D object.

#### 21.2.1.1 Wall AnalyticalModel

Every overridden NewWall method has a structural Boolean input parameter that indicates if the wall is structural in nature. Check for one of the following conditions:

- The wall family contains AnalyticalModel and the structural input parameter is true.
- WALL_ATTR_EXTERIOR is Foundation.

If either of these conditions applies, downcast the created Wall's AnalyticalModel property to AnalyticalModelWall. In this case, the StructuralUsage property of the wall will be Bearing. Without the AnalyticalModelWall, the StructuralUsage property is NonBearing.

#### 21.2.1.2 Floor AnalyticalModel

If the Floor parameter FLOOR_PARAM_IS_STRUCTURAL is true, the Floor AnalyticalModel is present and the Floor's AnalyticalModel can be downcast to an AnalyticalModelFloor.

**Note:** Use the Transaction method to get the newly created Element's AnalyticalModel. For example, after you create a wall using the NewWall method, the AnalyticalModel property returns null. Using the Transaction method, you can get both the AnalyticalModel and the wall geometry created by the API. For more details, refer to the Transaction chapter.

### 21.2.2 RigidLink

The AnalyticalModelFrame.RigidLink property only works for StructuralBeam when it links to StructuralColumn. The difference between the AnalyticalModelFrame properties Curve and Curves is that Curves includes Curve as well as the StructuralBeam RigidLink Curve if it is present.

**Note:** You cannot create a rigid link directly since it is not an independent object. You can create it using the manual command.

- StructuralColumn has the rigid link parameter (STRUCTURAL_ANALYTICAL_RIGID_LINK) with the ParameterType, YesNo.
- If this parameter is set to 1 and the StructuralBeam is in the proper location relative to the StructuralColumn, the rigid link is created automatically.

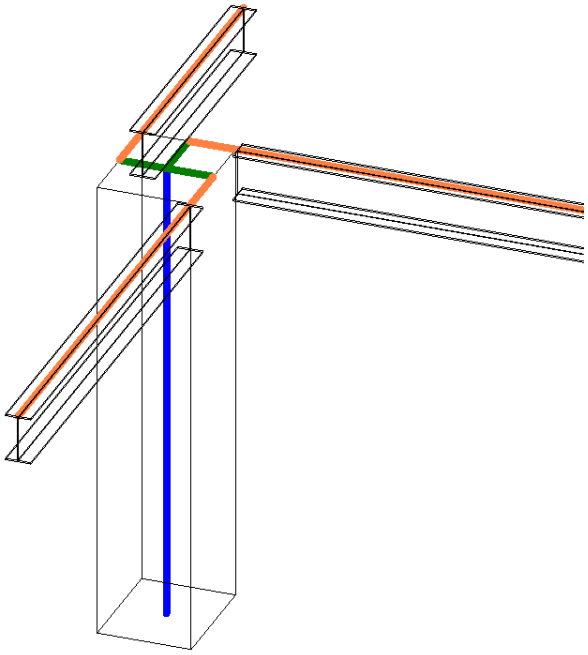


**Figure 219: RigidLink**

### 21.2.3 SupportData

SupportData provides information to support other Structural Elements. The following examples illustrate how to use the SupportData property in different conditions.

#### 21.2.3.1 Floor and StructuralBeam SupportData

When drawing a slab in sketch mode, select Pick Supports on the design bar. As shown in the following picture, a slab has three support beams. By iterating the slab SupportData InfoArray, you get the three Beams as well as the Line_Support SupportType.

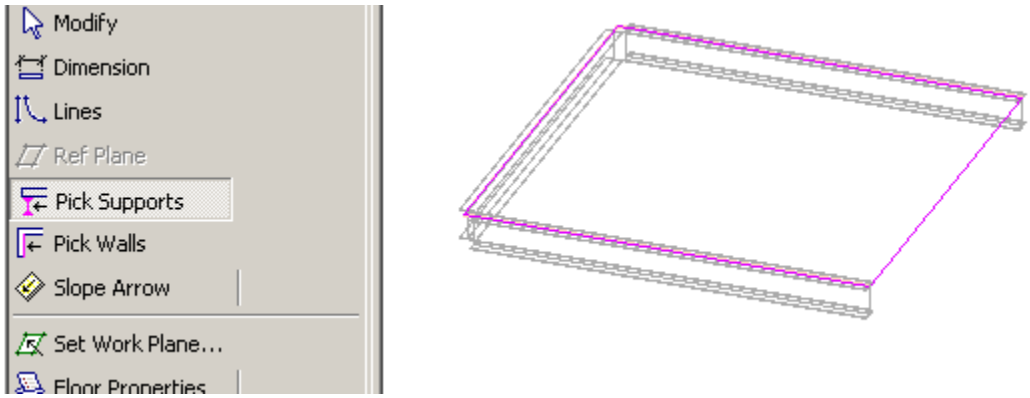



**Figure 220: Floor and StructuralBeam SupportData**

#### 21.2.3.2 Floor and Wall SupportData

After drawing a slab by picking walls as the support, you cannot get Walls from Floor SupportData. Instead, Floor is available in Wall SupportData. Get the support curve from the AnalyticalModelWall property Curves.

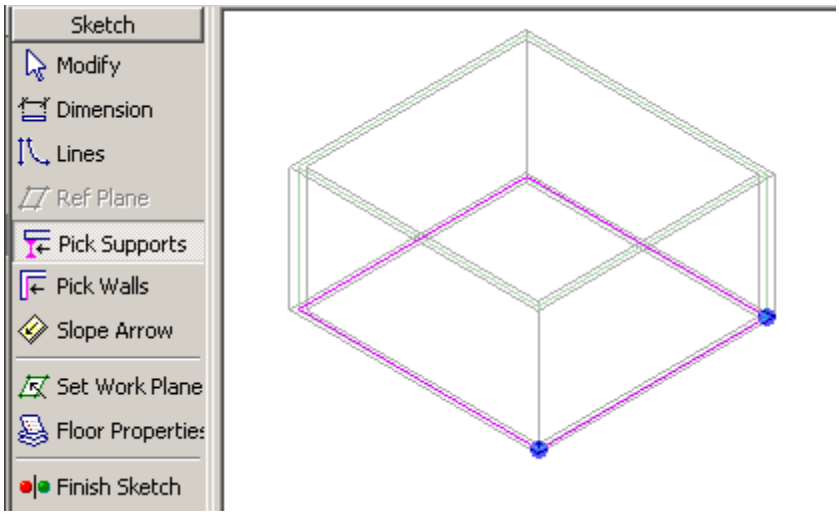



**Figure 221: Floor and Wall SupportData**

### 21.2.3.3 Structural Column, Beam and Brace SupportData

In the following picture, the horizontal beam has three Point_Supports--two StructuralColumns and one StructuralBrace. The brace has three Point_Supports-- two StructuralColumns and one StructuralBeam. Neither Column has a support Element.

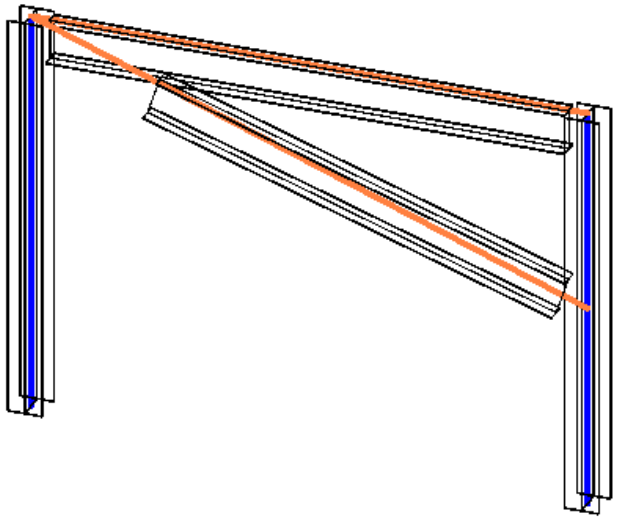


**Figure 222: StructuralElements SupportData**

### 21.2.3.4 BeamSystem and Wall SupportData

Though you can pick walls as supports when you draw a BeamSystem, its support information is not directly available because the BeamSystem does not have the AnalyticalModel property. The solution is to call the GetAllBeams method, to retrieve the AnalyticalSupportInfo for the Beams.

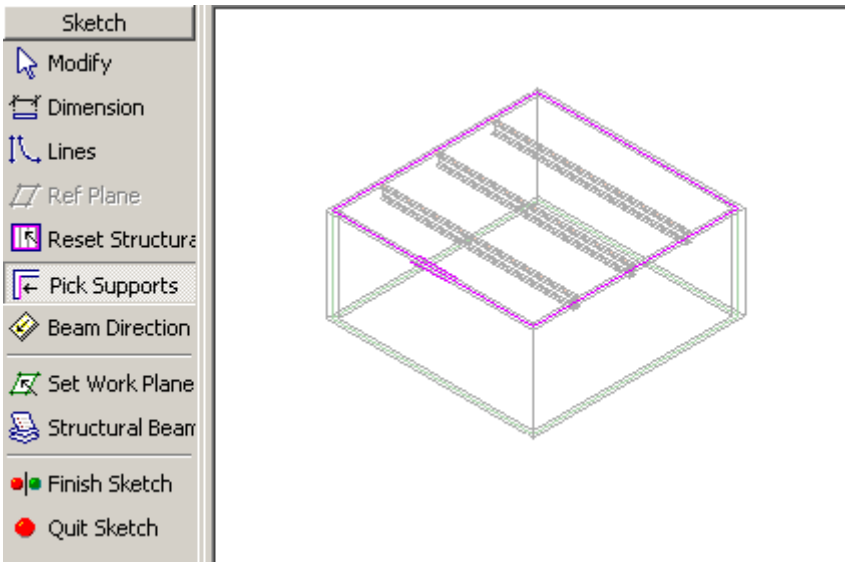



**Figure 223: BeamSystem and Wall SupportData**

### 21.2.3.5 ContFooting and Wall SupportData

For a Wall with a continuous Foundation, the Wall has a Line_Support with ContFooting available. The support curves are available using the AnalyticalModel3D.Curves. In the following sample, there are two Arcs in the Curve.

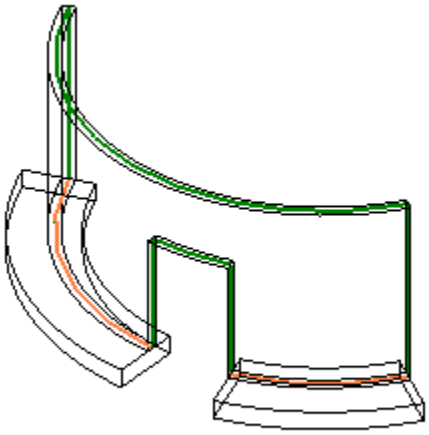


**Figure 224: ContFooting and Wall SupportData**

### 21.2.3.6 Isolated Foundation and StructuralColumn SupportData

StructuralColumns can have an Isolated Footing as a Point_Support. In this condition, the Footing can move with the supported StructuralColumn. The FamilyInstance with the OST_StructuralFoundation category is available from InfoArray. Generally, the support point is the bottom point of the AnalyticalModelFrame Curve property. It is also available after you get the FamilyInstance, Isolated Footing, and the AnalyticalModelLocation Point property.

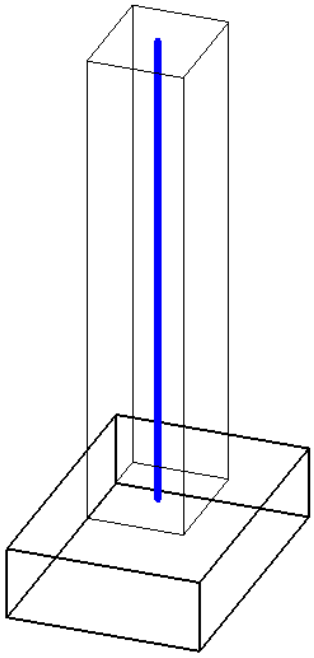


**Figure 225: Isolated Foundation (FamilyInstance) and StructuralColumn SupportData**

## 21.3 Loads

The following sections identify load settings and discuss load limitation guidelines.

### 21.3.1 Load Settings

All functionality on the Setting dialog box Load Cases and Load Combinations tabs can be rebuilt by the API. (Settings > Structural Settings > Load Cases tab and Load Combinations tab)

The following properties are available from the corresponding LoadCase BuiltInParameter:

| Property | BuiltInParameter |
|---|---|
| Case Number | LOAD_CASE _NUMBER |
| Nature | LOAD_CASE_NATURE |
| Category | LOAD_CASE_CATEGORY |

**Table 49 Load Case Properties and Parameters**

The LOAD_CASE_CATEGORY parameter returns an ElementId. The following table identifies the mapping between Category and ElementId Value.

| Load Case Category | BuiltInCategory |
|---|---|
| Dead Loads | OST_LoadCasesDead |
| Live Loads | OST_LoadCasesLive |
| Wind Loads | OST_LoadCasesWind |
| Snow Loads | OST_LoadCasesSnow |
| Roof Live Loads | OST_LoadCasesRoofLive |
| Accidental Loads | OST_LoadCasesAccidental |

| Load Case Category | BuiltInCategory |
|---|---|
| Temperature Loads | OST_LoadCasesTemperature |
| Seismic Loads | OST_LoadCasesSeismic |

**Table 50: Load Case Category**

The following Element subclasses create corresponding subclasses:

- NewLoadUsage creates LoadUsage
- NewLoadNature creates LoadNature
- NewLoadCase creates LoadCase
- NewLoadCombination creates LoadCombination.

Because they are all Element subclasses, they can be deleted using Document.Delete.

Be careful using the following input parameters.

- NewLoadCombination method
- LoadCaseArray factor
- LoadcombinationArray factor

```
public LoadCombination NewLoadCombination(string name,
   int typeInd, int stateInd, double[] factors, LoadCaseArray cases,
  LoadCombinationArray combinations, LoadUsageArray usages);
```

Factor size must be equal to or greater than the sum size of cases and combinations. For example,

- If cases.Size is M, combinations.Size is N,
- Factors.Size should not be less than M+N. The first M factors map to M cases in order, and the last N factors map to N combinations.
- Check that LoadCombination does not include itself.

There is no Duplicate method in the LoadCase and LoadNature classes. To implement this functionality, you must first create a new LoadCase (or LoadNature) object using the NewLoadCase (or NewLoadNature) method and then copy the corresponding properties and parameters from an existing LoadCase (or LoadNature).

### 21.3.2 Load Limitations

Currently, you cannot create LineLoad and AreaLoad with Host using the corresponding methods in Creation.Document. In other words, you cannot create a LineLoad along a StructuralBeam or an AreaLoad on a structural Wall. Even if the location of the LineLoad is created along the StructuralBeam's analytical model line, the Host property returns null.

Additionally, you can identify whether a Load has a host in the UI if it can be moved with the host element.

## 21.4 Analysis Link

With Revit Structure, an analytical model is automatically generated as you create the physical model. The analytical model is linked to structural analysis applications and the physical model is automatically updated from the results through the Revit Structure API. Some third-party software developers already provide bi-directional links to their structural analysis applications. These include the following:

- Robot Millennium from Robobat:(www.revit.robobat.com)
- RAM Structural System from RAM Intl (www.ramint.com/support/revit.jsp)
- RISA-3D and RISAFloor from RISA Technologies (www.risatech.com)
- Smart Modeller from CADS(www.cads.co.uk)
- Fastrak and S-Frame from CSC (www.cscworld.com)
- GSA from Oasys Software (Arup) (www.oasys-software.com/products Later in 2007)

The key to linking Revit Structure to other analysis applications is to set up the mapping relationship between the objects in different object models. That means the difficulty and level of the integration depends on the similarity between the two object models.

For example, during the product design process, design a table with at least the first two columns in the object mapping in the following table: one for Revit Structure API and the other for the structural analysis application, shown as follows:

| Revit Structural API | Analysis Application | Import to Revit |
|---|---|---|
| StructuralColumn | Column | NewStructuralColumn |
| Property: | | |
| ... | | |
| Location | | Read-only; |
| Parameter: | | |
| ... | | |
| Analyze as | | Editable; |
| AnalyticalModel: | | |
| ... | | |
| Profile | | Read-only; |
| RigidLink | | Read-only; |
| ... | | |
| Material: | | |
| ... | | |

**Table 51: Revit and Analysis Application Object Mapping**

# A Glossary

## A.1 API

The Application Programming Interface (API) defines how programs communicate with one another, much in the same way that a GUI (Graphical User Interface) is used by humans to communicate with applications.

## A.2 Array

Arrays hold a series of data elements, usually of the same size and data type. Individual elements are accessed by their position in the array. The position is provided by an index, which is also called a subscript. The index usually uses a consecutive range of integers, but the index can have any ordinal set of values.

## A.3 BIM

Building Information Modeling is the creation and use of coordinated, internally consistent, computable information about a building project in design and construction. In a BIM application the graphics are derived from the information and are not the original information itself like in general CAD applications.

## A.4 Class

In object-oriented programming (OOP), classes are used to group related Properties (variables) and Methods (functions) together. A typical class describes how those methods operate upon and manipulate the properties. Classes can be standalone or inherited from other classes. In the latter, a class from which others are derived is usually referred to as a Base Class.

## A.5 Events

Events are messages or functions that are called when an event occurs within an application. For example when a model is saved or opened.

## A.6 GUI

The Graphical User Interface (GUI) is the graphic interface that humans use to interact with computer programs. These are usually considered the windows, toolbars, and menus in a program.

## A.7 Iterator

An iterator is an object that allows a programmer to traverse through all elements in a collection (an array, a set, etc.), regardless of its specific implementation.

## A.8 Method

A method is a function or procedure that belongs to a class and operates or accesses the class data members. In procedural programming, this is called a function.

## A.9 Namespace

A namespace is an organizational unit used to group similar and/or functionally related classes together.

## A.10Overloading

Method overloading is when different methods (functions) of the same name are invoked with different types and/or numbers of parameters passed.

## A.11Properties

Properties are data members of a class accessible to the class user. In procedural programming this is called a variable. Some properties are read only (support Get() method) and some are modifiable (support Set() method).

## A.12Revit Families

A Family is a collection of objects called types. A family groups elements with a common set of parameters, identical use, and similar graphical representation. Different types in a family can have different values of some or all parameters, but the set of parameters - their names and their meaning - are the same.

## A.13Revit Parameters

There are a number of Revit parameter types.

- Shared Parameters can be thought of as user-defined variables.
- System Parameters are variables that are hard-coded in Revit.
- Family parameters are variables that are defined when a family is created or modified.

## A.14Revit Types

A Type is a member of a Family. Each Type has specific parameters that are constant for all instances of the Type that exist in your model; these are called Type Properties. Types have other parameters called Instance parameters, which can vary in your model.

## A.15Sets

A set is a collection (container) of values without a particular order and no repeated values. It corresponds with the mathematical concept of set except for the restriction that it has to be finite.

## A.16Element ID

Each element has a corresponding ID. It is identified by an integer value. It provides a way of uniquely identifying an Element within an Autodesk Revit project. It is only unique for one project, but not unique across separate Autodesk Revit projects.

## A.17Element UID

Each element has a corresponding UID. It is a string identifier that is universally unique. That means it is unique across separate Autodesk Revit projects.

# B FAQ

## B.1 General Questions

**Q:** How do I reference an element in Revit?

**A:** Each element has an ID. The ID that is unique in the model is used to make sure that you are referring to the same element across multiple sessions of Revit.

**Q:** Can a model only use one shared parameter file?

**A:** Shared parameter files are used to hold bits of information about the parameter. The most important piece of information is the GUID (Globally Unique Identifier) that is used to insure the uniqueness of a parameter in a single file and across multiple models.

Revit can work with multiple shared parameter files but you can only read parameters from one file at a time. It is then up to you to choose the same shared parameter file for all models or a different one for each model.

In addition, your API application should avoid interfering with the user's parameter file. Ship your application with its own parameter file containing your parameters. To load the parameter(s) into a Revit file:

- The application must remember the user parameter file name.
- Switch to the application's parameter file and load the parameter.
- Then switch back to the user's file.

**Q:** Do I need to distribute the shared parameters file with the model so other programs can use the shared parameters?

**A:** No. The shared parameters file is only used to load shared parameters. After they are loaded the file is no longer needed for that model.

**Q:** Are shared parameter values copied when the corresponding element is copied?

**A:** Yes. If you have a shared parameter that holds the unique ID for an element in your database, append the Revit element Unique ID or add another shared parameter with the Revit element unique ID. Do this so that you can check it and make sure you are working with the original element ID and not a copy.

**Q:** Are element Unique IDs (UID) universally unique and can they ever change?

**A:** The element UIDs are universally unique, but element IDs are only unique within a model. For example, if you copy a wall from one Revit project to another one, the UID of the wall is certain to change to maintain universal uniqueness, but the ID of the wall may not change.

**Q:** Revit takes a long time to update when my application sends data back to the model. What do I need to do to speed it up?

**A:** You can try using the SuspendUpdating command. See the FrameBuilder example in the SDK.

**Q:** What do I do if I want to add shared parameters to elements that do not have the ability to have shared parameters bound to them? For example, Grids or Materials.

**A:** If an element type does not have the ability to add shared parameters, you need to add a project parameter. This does make it a bit more complicated when it is time to access the shared parameter associated with the element because it does not show up as part of the element's parameter list. By using tricks like making the project shared parameter a string and including the element ID in the shared parameter you can associate the data with an element by first parsing the string.

**Q:** How do I access the saved models and content BMP?

**A:** The Preview.dll is a shell plugin which is an object that implements the IExtractImage interface. IExtractImage is an interface used by the Windows Shell Folders to extract the images for a known file type.

For more information, review the information at http://windowssdk.msdn.microsoft.com/en-us/library/ms645964.aspx

CRevitPreviewExtractor implements standard API functions:

```
STDMETHOD(GetLocation)(LPWSTR pszPathBuffer,
                       DWORD cchMax,
                       DWORD *pdwPriority,
                       const SIZE *prgSize,
                       DWORD dwRecClrDepth,
                       DWORD *pdwFlags);
STDMETHOD(Extract)(HBITMAP*);
```

It registers itself in the registry.

## B.2 Revit Structure Questions

**Q:** Sometimes the default end releases of structural elements render the model unstable.

**A:** The Analytical Model Check feature introduced in Revit Structure R3 can find some of these issues. When importing the analytical model, you are asked if you want to retain the release conditions from RST (Revit Structure) or if you want to set all beams and columns to fixed. When re-importing the model to RST, always update the end releases and do not overwrite the end releases on subsequent export to analysis programs.

**Q:** I am rotating the beam orientation so they are rotated in the weak direction. For example, the I of a W14X30 is rotated to look like an H by a 90 degree rotation. How is that rotation angle accessed in the API?

Because the location is a LocationCurve not a LocationPoint I do not have access to the Rotation value so what is it I need to check? I have a FamilyInstance element to check so what do I do with it?

**A:** Take a look at the RotateFramingObject example in the SDK. It has examples of how to get and change the beam braces and columns rotation angle.

**Q:** How do I add new concrete beam and column sizes to a model?

**A:** Take a look at the FrameBuilder sample code in the SDK

**Q:** How do I view the true deck layer?

**A:** There is an example in the SDK called DeckProperties that provides information about how to get the layer information for the deck. The deck information is reported in exactly the same way as it is in the UI. The deck dimension parameters are shown as follows.

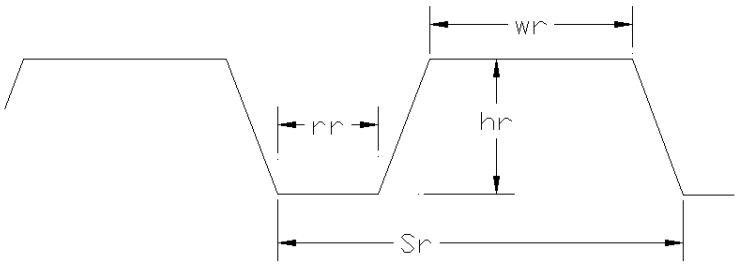



**Figure 226: Deck dimension parameters**

**Q:** How do I tell when I have a beam with a cantilever?

**A:** There is no direct way in the Revit database to tell if a beam has a cantilever. However, one or more of the following options can give you a good guess at whether a section is a cantilever:

1. There are two parameters called Moment Connection Start and Moment Connection End. If the value set for these two is not None then you should look and see if there is a beam that is co-linear and also has the value set to something other than None. Also ask the user to make sure to select Cantilever Moment option rather than Moment Frame option.
2. Trace the connectivity back beyond the element approximately one or two elements.
3. Look at element release conditions.

**Q:** How do I model a foundation with end overhangs under a wall?

**A:** The wall foundation capability in Revit Structure does not support an overhang at its ends. The best solution is to use the foundation slab feature in the GUI or API. You can create them using the Autodesk.Revit.Create.NewFloor methods. For the STRUCTURAL_FLOOR_ANALYZES_AS, you can use AA_Mat or AA_SlabOnGrade for foundation or AA_Slab if it is a floor.

**Q:** When exporting a model containing groups to an external program, the user receives the following error at the end of the export:

"Changes to group "Group 1" are allowed only in group edit mode. Use the Edit Group command to make the change to all instances of the group. You may use the "Ungroup" option to proceed with this change by ungrouping the changed group instances."

**A:** Currently the API does not permit changes to group members. You can programmatically ungroup, make the change, regroup and then swap the other instances of the old group to the new group to get the same effect.

# C Hello World for VB.NET

Directions for creating the sample application for Visual Basic .NET are available in the following sections. The sample application was created using Microsoft Visual Studio.

## C.1 Create a New Project

The first step in writing a VB.NET program with Visual Studio is to choose a project type and create a new project.

1. From the File menu, select New> Project….
2. In the Project types frame, click Visual Basic.
3. In the Templates frame, click Class Library. The application assumes that your project location is: D:\Sample.
4. In the Name field, type HelloWorld as the project name.
5. Click OK.

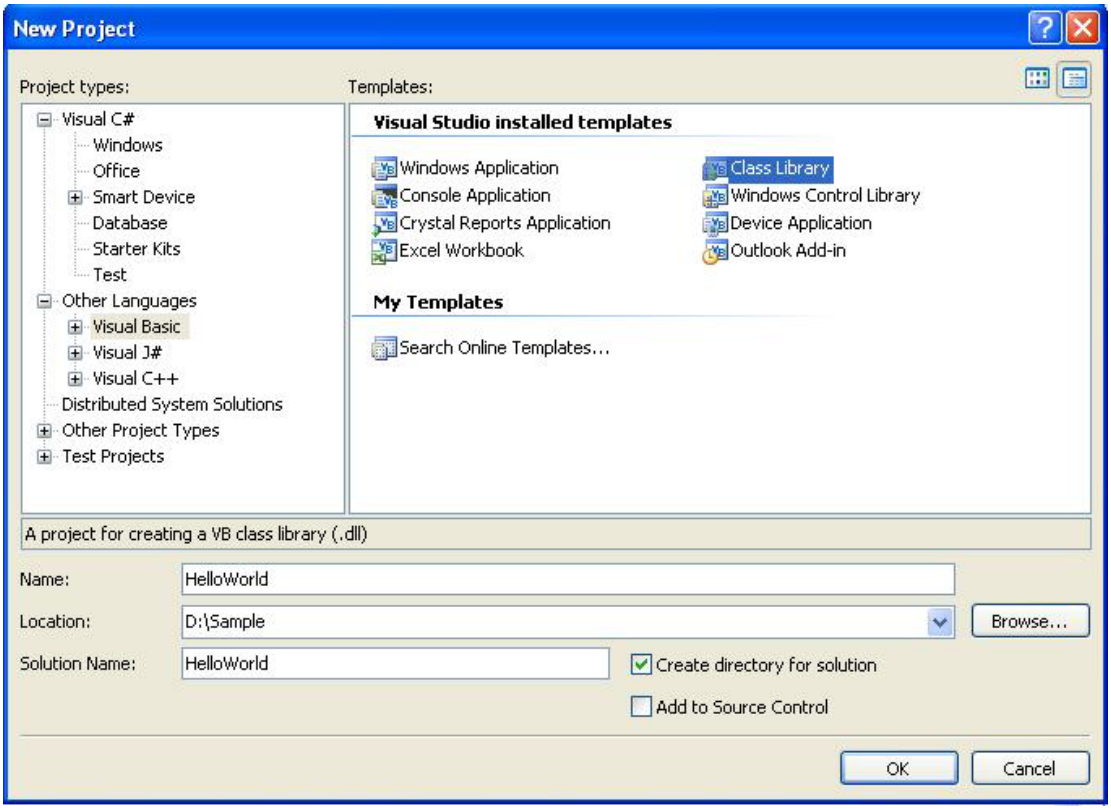



**Figure 227: New Project dialog box**

## C.2 Add Reference and Namespace

VB.NET uses a process similar to C#. After you create the Hello World project, complete the following steps:

1. Right-click the project name in the Solution Explorer to display a context menu.
2. From the context menu, select Properties to open the Properties dialog box.
3. In the Properties dialog box, click the References tab. A list of references and namespaces appears.

4. Click the Add button to open the Add Reference dialog box.
5. In the Add Reference dialog box, click the Browse tab. Locate the folder where Revit is installed and click the RevitAPI.dll. For example the installed folder location might be C:\Program Files\Revit Architecture 2008\Program\RevitAPI.dll.
6. Click OK to add the reference and close the dialog box.

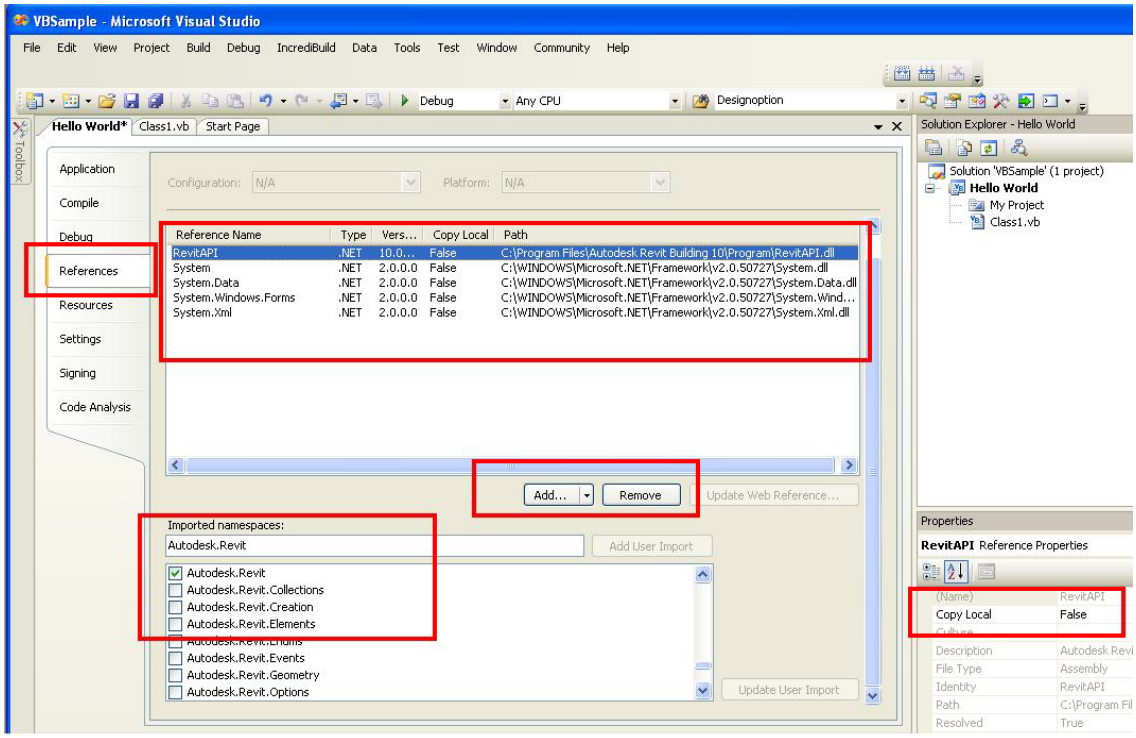



**Figure 228: Add references and import Namespaces**

After adding the reference, you must import the namespaces used in the project. For this example, import the Autodesk.Revit namespace

To complete the process, click RevtAPI in the Reference frame to highlight it. Set Copy Local to False in the property frame.

## C.3 Change the Class Name

To change the class name, complete the following steps:

1. In the Solution Explorer, right-click Class1.vb to display a context menu.
2. From the context menu, select Rename. Rename the file HelloWorld.vb.
3. In the Solution Explorer, double-click HelloWorld.vb to open it for editing.

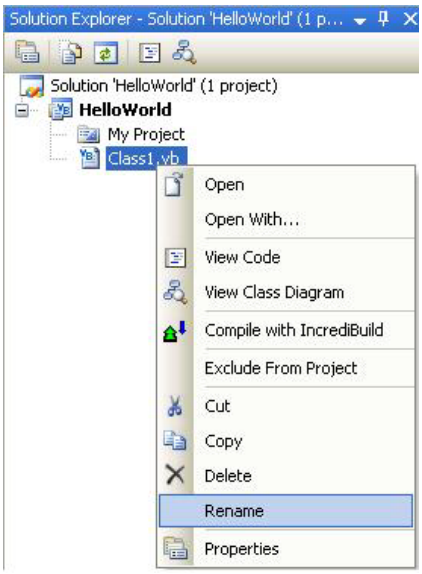



**Figure 229: Change the class name**

## C.4 Add Code

When writing the code in VB.NET, you must pay attention to key letter capitalization.

**Code Region 5-Getting Start : HelloWorld VB**

```
Imports System
Imports System.Windows.Forms

Imports Autodesk.Revit



Public Class HelloWorld
    Implements Autodesk.Revit.IExternalCommand

    Public Function Execute(ByVal commandData As Autodesk.Revit.ExternalCommandData, _
                            ByRef message As String, ByVal elements As
Autodesk.Revit.ElementSet) As Autodesk.Revit.IExternalCommand.Result _
      Implements Autodesk.Revit.IExternalCommand.Execute

        MsgBox("Hello World")
        Return IExternalCommand.Result.Succeeded

    End Function

End Class
```

## C.5 Modify the Revit.ini File

After you add the code, you must build the file. The HelloWorld.dll file appears in the project output directory. If you want to invoke the application in Revit, modify the Revit.ini file to register it.

1. To edit the Revit.ini file, open it for editing in Notepad. The Revit.ini file is usually located in the Revit installation directory on your computer. For example: C:\Program Files\Revit Architecture 2008\Program.
2. Add the following to the end of the existing code:

```
[ExternalCommands]
ECCount=1
ECClassName1= HelloWorld.HelloWorld
ECAssembly1= D:\Sample\HelloWorld\bin\Debug\HelloWorld.dll
ECName1= HelloWorld
ECDescription1=Implementation of HelloWorld within Autodesk Revit
```

**Note:** ECAssembly1 is the path to the assembly, D:\Sample\HelloWorld\bin\Debug\HelloWorld.dll in this particular example.

Refer to the Add-In Integration chapter for more details about the Revit.ini file.

## C.6 Build the Program

After completing the code, you must build the file. From the Build menu, click Build Solution. Output from the build appears in the Output window indicating that the project compiled without errors.

## C.7 Debug the Program

Running a program in Debug mode uses breakpoints to pause the program so that you can examine the state of variables and objects. If there is an error, you can check the variables as the program runs to deduce why the value is not what you might expect.

1. In the Solution Explorer window, right-click the HelloWorld project to display a context menu.
2. From the context menu, click Properties. The Properties window appears.
3. Click the Debug tab.
4. In the Debug window Start Action section, click Start external program and browse to the Revit.exe file. By default, the file is located at the following path, C:\Program File\Revit Architecture 2008\Program\ Revit.exe.

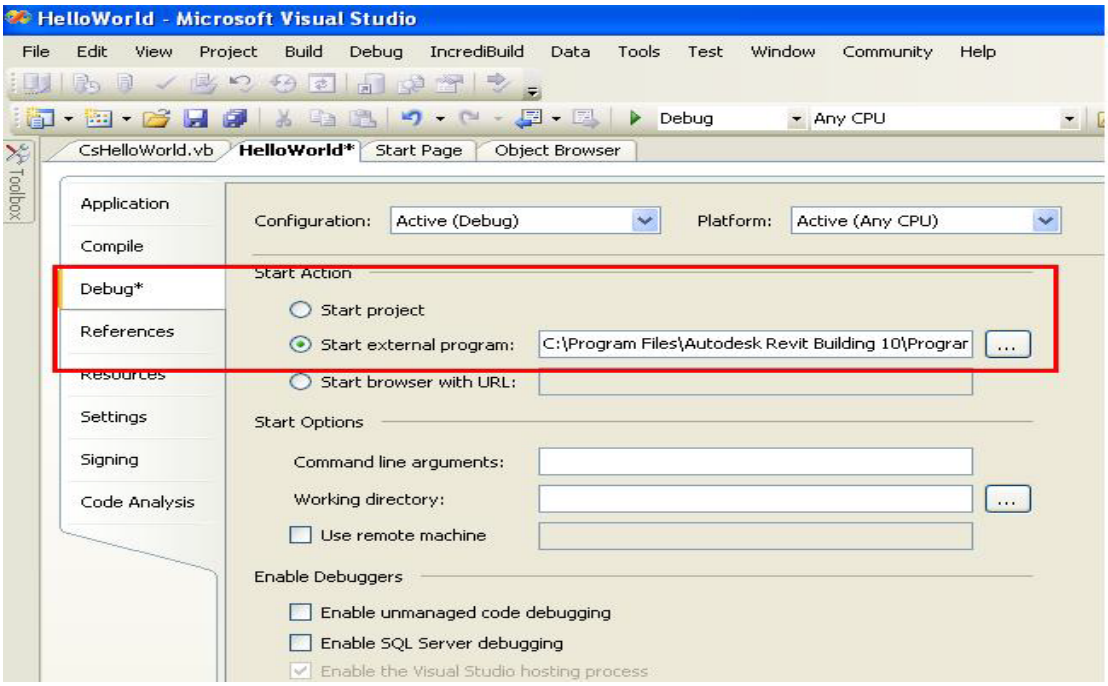



**Figure 230: Set Debug environment**

5. From the **Debug** menu, select **Toggle Breakpoint** (or press F9) to set a breakpoint on the following line.

```
MsgBox("Hello World")
```

6. Press F5 to start the debug procedure.
7. Test the debugging
   - From the Revit menu bar, click Tools > External Tools. HelloWorld appears in the submenu.

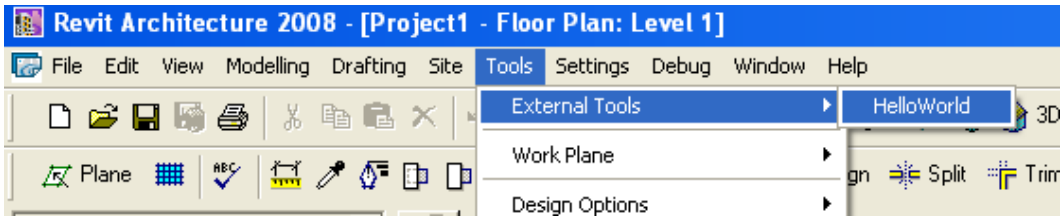



**Figure 231: HelloWorld External Tools command**

   - Click HelloWorld to execute the program, activating the breakpoint.
   - Press F5 to continue executing the program. The following system message appears.

**Figure 232: System message**